世界是数字的

[美] Brian W. Kernighan 著 李松峰 徐建刚 译



盖茨和**扎克伯格**导师**刘易斯**
谷歌常务董事长**施密特**
联袂推荐

面对这个时代最伟大的技术，无论你是谁，都很难置身事外
请跟随他，请跟随普林斯顿大学，开启一次华美的计算机之旅



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



Brian W. Kernighan

世界顶尖计算机科学家，曾为贝尔实验室计算科学研究中心高级研究人员，现任普林斯顿大学教授。他是AWK语言和AMPL语言的发明人，还参与过UNIX和许多其他系统的开发，同时出版了*The C Programming Language*、*The Practice of Programming*、*The Elements of Programming Style*等在计算机领域影响深远的著作。

从1999年开始，作者在普林斯顿大学开设了一门名叫“我们世界中的计算机”的课程（COS 109：Computers in Our World），这门课是向非计算机专业的学生介绍计算机基本常识的，多年来大受学生追捧。除了向学生讲解计算机理论知识，这门课还有相应的实验课——学生可以试着用流行的编程语言写几行代码，大家一起讨论苹果、谷歌和微软的技术如何渗入日常生活的每个角落。本书就是以这门课程的讲义为主要内容重新编写而成，它解释了计算机和通信系统的工作原理，并讨论了新技术带来的社会、政治和法律问题。

Kernighan主张所有人都应该了解计算机，在他心目中，此书相当于“写给未来总统的计算机读物”，是新一代公民必读之作。本书内容甚至会影响我们在微博和社区上的活动，以及使用智能手机的方式。



中文版序

如今，计算机、通信系统，以及由它们支撑的数字产品已经无所不在了！笔记本电脑、手机和互联网，这些都是显而易见的。更多的则是我们平常看不见的，比如那些栖身于电子设备、汽车、火车、飞机、电力系统、医疗设备中的计算机。偶尔，透过某些提示，你会得知世界上有无数系统正在悄悄收集、分享你的个人信息，这些信息甚至会被用在违背你意愿的地方。

《世界是数字的》简明扼要但又深入全面地解释了计算机和通信系统背后的秘密，旨在让没有技术背景的读者更好地理解自己生活的这个数字世界。这本书解释了如今计算和通信的运作方式，包括硬件、软件、互联网，还有万维网，同时还探讨了新技术引发的社会、政治和法律问题，让你明白现实当中的一些难题和迫不得已的折中。

我相信，了解这些技术常识对于任何人都非常重要，无论你是什么背景。非常高兴这本书由李松峰和徐建刚翻译成中文出版，希望中国读者能够喜欢。

Brian Kernighan



作者在普林斯顿大学的办公室，背后桌上放的是本书英文版[①]

① 本书插页所有照片均为作者 Brian Kernighan 提供，是他个人在普林斯顿大学校园拍摄的。——编者注

Foreword to the Chinese Translation of *D is for Digital*

Computers and communications systems and the many things enabled by them are all around us. Some of this is highly visible, like laptop computers, mobile phones and the Internet; much is invisible, like the computers in electronic equipment, cars, trains, planes, power systems and medical equipment. Occasionally we get disturbing hints about the myriad systems that quietly collect, share and sometimes misuse personal data about us.

“D is for Digital” is a compact but detailed and thorough explanation of how computers and communications systems work, for non-technical readers who want to better understand the world they live in. It explains how today’s computing and communications world operates, from hardware through software to the Internet and the web. The social, political and legal issues that new technology creates are discussed as well, so you can understand the difficult issues we face and appreciate the tradeoffs that have to be made to resolve them.

I believe that it is important that people everywhere, no matter what their background, understand this amazing pervasive technology. I am absolutely delighted that this Chinese translation by Li Songfeng and Xu Jiangang is available for interested readers in China, and I hope that you will enjoy reading it.

Brian Kernighan



作者在普林斯顿大学绿树环抱的拿骚楼（Nassau Hall）前。拿骚楼始建于 1756 年，是普林斯顿大学最古老的建筑



作者在普林斯顿大学计算机科学楼门厅。该校计算机科学专业在世界大学计算机专业排名中名列前茅，师资队伍中不仅有图灵奖得主，还有多名计算机行业内泰斗级专家教授，Kernighan 便是其中之一


普林斯顿大学本德海姆金融中心（Bendheim Center for Finance）秋色。本德海姆金融中心成立于 1998 年，鼓励从定量或数学角度跨学科研究金融



普林斯顿大学研究生院。其中有四个尖塔的是克利夫兰塔（Cleveland Tower），它是普林斯顿大学标志建筑之一，1913 年为纪念该校理事、美国前总统格罗弗·克利夫兰而建，是该校一系列学院歌特建筑中的代表



隔着校内高尔夫球场远眺研究生院

图书在版编目（ＣＩＰ）数据

世界是数字的 / (美) 柯林汉 (Kernighan,B.W.) 著；
李松峰，徐建刚译. -- 北京 : 人民邮电出版社，2013.6
书名原文: D is for digital
ISBN 978-7-115-31875-6

Ⅰ. ①世… Ⅱ. ①柯… ②李… ③徐… Ⅲ. ①电子计
算机－普及读物 Ⅳ. ①TP3-49

中国版本图书馆CIP数据核字(2013)第115306号

内 容 提 要

家用电器、汽车、飞机、相机、手机、GPS 导航仪，还有游戏机，虽然你看不见，但这些设备都有计算能力。手机通信网络、有线电视网络、空中交通管制系统、电力系统、银行和金融服务系统等基础设施背后无一不是计算机在支撑。如今的世界是数字的，而计算机和计算无处不在。这本书就是要告诉大家数字世界有关计算机的一切。本书没有高深莫测的专业术语，但它全面解释了当今计算和通信领域的工作方式，包括硬件、软件、互联网、通信和数据安全，并且讨论了新技术带来的社会、政治和法律问题。

无论你有没有计算机背景，无论你从事什么职业，只要你认同自己生活在数字时代，这本书就是必读的！

◆ 著　　　[美] Brian W. Kernighan
译　　　李松峰　徐建刚
责任编辑　刘美英
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：720×960　1/16
印张：17　　彩插：2
字数：302　　2013年6月第1版
印数：1－4 000册　　2013年6月北京第1次印刷
著作权合同登记号　图字：01-2012-3281号

定价：49.00元
读者服务热线：(010)51095186转604　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

献给梅格

Brian W. Kernighan的其他著作

The Elements of Programming Style（与 P. J. Plauger 合著）

Software Tools（与 P. J. Plauger 合著）

Software Tools in Pascal（与 P. J. Plauger 合著）

The C Programming Language（与 D. M. Ritchie 合著）

The AWK Programming Language（与 A. V. Aho 和 P. J. Weinberger 合著）

The Unix Programming Environment（与 R. Pike 合著）

AMPL: A Modeling Language for Mathematical Programming（与 R. Fourer 和 D. M. Gay 合著）

The Practice of Programming（与 R. Pike 合著）

译者序

普林斯顿在哪儿？在纽约和费城之间。如果这么说不足以让你确定这个小镇的方位，那我们换一种说法：普林斯顿在美国东部的位置，大致相当于中国北京到上海之间的徐州。徐州古称“彭城”，是刘邦故里，项羽故都。普林斯顿呢，则是美国首屈一指的普林斯顿大学所在地。

很多人对普林斯顿大学的认知来自获得奥斯卡金像奖的电影《美丽心灵》。这部电影以 1994 年获得诺贝尔经济学奖的数学家小约翰·福布斯·纳什（Jr.John Forbes Nash）为原型。1950 年代，20 出头的纳什在普林斯顿攻读博士期间，发表了一篇关于非合作博弈的博士论文，确立了他博弈论大师的地位。而同时代的普林斯顿可谓大师云集，爱因斯坦、冯·诺依曼、列夫谢茨（数学系主任）、阿尔伯特·塔克、阿伦佐·切奇、哈罗德·库恩……都在这里。普林斯顿大学直到今天的在校学生也不过 7000 多人，但却人才辈出：两位美国总统、44 位州长、33 位诺贝尔奖得主。

1999 年秋天至今，普林斯顿大学计算机科学系开设了一门名为“我们世界中的计算机”（Computers in our World）的课程。这门课是向非计算机专业的学生介绍计算机、互联网和通信方面基本常识的。而开设并讲授这门课程的教授 Brian Kernighan，曾在贝尔实验室工作 30 多年，与 Unix 系统的创造者 Ken Thompson 和 Dennis Ritchie（C 语言发明人）是同事，对 Unix 系统也做出了突出贡献。他与 Dennis Ritchie 合著的 *The C Programming Language* 是世界上第一本被广泛认可的 C 语言教程，被称为“K&R C”。他还发明了 AWK 和 AMPL 编程语言。

今天，计算机和互联网已经无处不在，已经像水、电、气一样，成为我们日常生活须臾不可缺少的必需品。了解这些对我们工作、学习和生活正产生巨大影响的基础设施的工作原理和运行机制，知道信息技术在给我们带来巨大便利的同时，还带来了哪些烦恼、隐忧，甚至前所未有的对我们人身、财产的各种威胁，我们才能真正融入这个时代，在各个领域立于不败之地。

普林斯顿大学的学生何其有幸，能够亲耳聆听顶尖计算机科学家的启蒙和常识普及课。而现在，作者把自己多年教学的讲义整理成这么一本集知识性、趣味性和科普性于一体的读物，让我们“到普林斯顿大学听计算机讲座”的梦想变成了现实。

这本书由李松峰和徐建刚合作翻译，其中李松峰翻译了“致读者”、第 2 章到第 8 章和第 11 章、第 12 章，以及书后的注解和词汇表，徐建刚翻译了“开篇语”、第 1 章、第 9 章和第 10 章。感谢李琳骁在翻译期间审读本书，并提出了很多修改意见，让表达更加通顺流畅。感谢图灵社区热心读者，特别是 xslidian 在阅读本书电子版期间提出的勘误，帮助我们更正了不少翻译错误。

致读者

1999 年秋天至今，我一直在普林斯顿大学教一门课，叫“我们世界中的计算机”。这门课的名字实在是含糊得可以，说起来都让人尴尬。这是我当时用了不到五分钟想出来的，结果“一失足成千古恨”，再想改就难了。不过，教这门课倒是给了我极大的乐趣，让我的工作几乎变成了享受。

开这门课主要是因为我们身边的计算机和与计算相关的东西已经随处可见了。每个学生都有一台计算机，其中任何一台的计算能力都比 1964 年我本科毕业时，普林斯顿大学那台造价数百万美元、占据一个很大房间的计算机强大得多。今天的一部手机甚至都比那时候的一台计算机处理能力强。每个学生都可以高速上网，美国至少半数以上的人也一样。我们搜索、上网购物，使用电子邮件、短信和社交网站与亲朋好友保持联络。

但这只是计算机时代的一个小小的缩影，不为人知的部分就像隐藏在海平面之下的冰山一样巨大。因为看不到，所以我们并不觉得家用电器、汽车、飞机，以及无所不在、司空见惯的电子设备——照相机、摄像机、游戏机、DVD 播放机、GPS 导航仪中都隐藏着计算机。

不经意间，对这些设备品头论足的信息也会进入我们的视野。就像有一次某报道引用了惠普一位高层领导的话：“本质上，数码相机就是一台带镜头的计算机。”同一篇报道中也引用了一位用户的话，这位用户好像不怎么高兴：“这哪是相机啊，根本就是一台计算机！”这是在抱怨有时候使用计算机也不容易。

而且，电话网络、电视和有线网络、空中交通管制、电网，还有银行和金融服务等公共设施对计算机的依赖程度也超出了我们的想象。

无所不在的计算机处处以意想不到的方式影响着我们。尽管我们时不时听说监控系统的规模不断在扩大，人们的隐私受到了侵害，电子投票带来了风险，但恐怕谁也没有意识到这些问题的根源就在于计算和通信的发展。

我们中的绝大多数人都不会与这些系统直接扯上关系，但每个人都会受到它们极大的影响，有些人则必须因它们而做出重大抉择。要是每个人对计算机都能多一点了解会不会更好呢？一个受过教育的人至少应该知道计算机的基本常识：计算机能做什么，怎么做；它们不可能做什么，而什么只是眼下还做不了；它们之间如何通信，通信过程中会发生什么；还有计算机和通信给我们生活的这个世界带来了哪些影响。

听我课的学生一般没有工科背景，他们专业学的都不是工程、物理或者数学。大部分学生主修的都是英语和政治、历史、古典文学、经济、音乐和艺术，几乎涵盖了人文和社会科学的各个专业。上完我的课之后，这些才华横溢的年轻人可以毫无障碍地阅读并理解计算机相关的文章和新闻报道，而且能够从中汲取更多的知识，甚至还可以指摘几处不够专业的表述。总的来说，我希望自己的学生对技术抱有理智批判的态度，知道它好，但它也不是万灵丹；反之，尽管技术有时会带来负面的结果，但它也并非一无是处。

理查德·穆勒（Richard Muller）的那本《未来总统的物理课》解释了作为国家领导人应该知道的科学和技术造成的社会问题，比如核威胁、恐怖主义、能源、全球变暖，等等。不想当总统但生活在信息时代的公民也应该很好地了解这些问题。虽然科学原理与推理论证有很多不同，但穆勒却能够很好聚焦于每个主题，聚焦于每个人都应该了解的物理常识。他的写作手法启发我让这本书成为“未来总统的计算机与通信”。作为总统应该了解哪些计算机和通信知识？一位信息时代的公民应该知道哪些？虽然每个人心目中的想法不一样，但这本书是我给出的答案。

这本书涵盖了三个核心技术领域：硬件、软件和通信。整本书都围绕这三个主题展开。

硬件是看得见摸得着的。不管是在家里，还是在办公室，计算机都是我们可以看到，可以触摸的。当然，还有我们每天随身携带的手机。计算机的内部都有什么，它是怎

么运转的，是根据什么原理制造的？它怎么保存和处理信息？什么是比特，什么是字节？怎么用它们来表示音乐、电影，还有一切？

*软件*是告诉计算机做什么的指令，几乎看不见，摸不着。通过计算可以做什么，计算速度可以有多快？怎么告诉计算机做什么？为什么让软件不出错很难？为什么它们有时候很难用？

*通信*就是计算机、手机和其他设备之间为了我们的需要而进行的对话，同时也让我们人和人之间能够交流，涉及互联网、万维网、电子邮件、社交网络等多种途径。这些东西的工作原理是什么？它们的好处显而易见，但有什么风险吗？特别是隐私和安全方面，该怎么解决呢？

在这三个主题之外，人们通常都会想到*数据*。数据指的是通过硬件及软件收集、存储和处理的，以及通信系统传送到世界各地的全部信息。其中部分数据是自愿公开的，主要是用户上传的照片和视频，有率性而为、不顾及后果的，也有时时警惕、谨小慎微的。还有一些是我们个人的信息，通常是在我们并不知情的情况下被收集和共享，根本没得商量。

无论你是总统，还是平民百姓，都应该了解这个计算机世界，因为它对每个人都有切身影响。无论工作和生活与技术距离有多遥远，你总有机会接触技术和搞技术的人。了解一些计算机和通信的常识都将对你大有助益，最低限度也能让你知道推销人员或服务热线什么时候向你隐瞒了事实。没错，无知有害。假如你不知道病毒、网络钓鱼以及类似的风险是怎么回事，那你受害的机率一定会大大增加；假如你不知道社交网络怎么泄露甚至任意传播你认为是个人隐私的信息，那你无意间泄露的很可能比自己想象的还要多；假如你对商业利益集团不顾一切从你的个人信息中挖掘线索这件事毫不知情，那你就会为了蝇头小利而出卖自己的隐私；假如你不知道在咖啡店和飞机上使用个人银行服务是有风险的，那么你的钱和身份就会让网络窃贼有可乘之机。

是要把这本书写成一本带习题的教材，每章后面再附上涉及各个知识点和实例的思考题呢，还是写成一本出版社称之为“科普书”的大众读物，字里行间点缀一些奇闻轶事，让它看起来根本不像课本呢？结果是两种倾向都有体现。不管怎样，这肯定不是一本带脚注、参考文献，句子都很长的“学术著作”。这种书连我这样的作者都怕得

要死，更何况可怜的读者。本书没有脚注，参考资料也不敢说没有遗漏。其中任何一个主题都可以大大地扩展，可要是每个主题都讲得很细，恐怕这本书的篇幅还要多出10倍，而且用处也没那么大了。本书末尾推荐阅读部分给出了一些我特别喜欢的书，还有一些网站链接，可以作为本书的补充阅读资料。

我希望读者可以从头到尾地阅读这本书，但你也可以先挑一些自己感兴趣的内容开始读，然后再看其他章节。举个例子，你可以先从第8章开始，依次了解计算机网络、手机、互联网、万维网和隐私等话题。虽然其中个别地方需要先弄明白前面几章的内容，但大多数内容还是可以直接看懂的。那些涉及数学的内容，比如讲二进制的第2章可以都先不看。有几章讲编程语言细节部分的内容，不想看也可以跳过。本书末尾有一个术语表，给出了一些重要术语和缩写的定义或解释。

讲计算机的书都很容易过时，当你读到这本书时，肯定会发现其中有些资料已经不那么准确，或者至少是过时了。不过，对于那些长期都有价值的东西，我也尽力清晰地传达给读者了。至于其他内容，比如更新、勘误或补充资料之类的，请读者拨冗访问一下本书网站 kernighan.com。

至于我写这本书的目标，是希望读者能对计算机和通信技术有一个深入的了解，真正明白它们的工作原理，它们的起源，还有未来的发展趋势。然后，能够从对自己有益的角度重新看待这个世界。果能如此，吾愿足矣。

致谢

首先，对朋友和同事们给予我无私的帮助和建议深表谢意。这里面要特别感谢Jon Bentley，他给每一页草稿都做了细致的批注。感谢Clay Bavor、Dan Bentley、Hildo Biersma、Stu Feldman、Gerard Holzmann、Joshua Katz、Mark Kernighan、Meg Kernighan、Paul Kernighan、David Malan、Tali Moreshet、Jon Riecke、Mike Shih、Bjarne Stroustrup、Howard Trickey和John Wait极其认真地审读全书，提出很多好建议， 让我避免了一些重大失误。还要感谢Jennifer Chen、Doug Clark、Steve Elgersma、Avi Flamholz、Henry Leitner、Michael Li、Hugh Lynch、Patrick McCormick、Jacqueline Mislow、Jonathan

Rochelle、Corey Thompson，以及Chris Van Wyk给出宝贵的评注。但愿他们一眼就看出我在哪里遵从了他们的建议，而不会留意那几处我没听劝的地方。

David Brailsford 根据自己来之不易的经验给了我很多有用的建议，有出版方面的，也有文字排版上的。Greg Doench 和 Greg Wilson 也毫无保留地给了我一些出版建议。感谢 Gerard Holzmann 和 John Wait 提供照片。

Harry Lewis 是 2010~2011 学年我在哈佛时负责接待我的人，本书的前几稿就是在那儿写的。Harry 的建议，还有他讲授类似课程的经验，对我帮助很大，他给我的几份草稿写的评注也一样。哈佛的工程和应用科学学院、伯克曼互联网与社会研究中心为我提供了办公的地方，各种设施，还有融洽友好催人奋进的环境，以及每天一顿免费的午餐（世上真有免费午餐！）。

最后，我特别感谢选修“COS 109：Computers in our World”这门课的几百位学生。他们的关注、热情和友谊一直都是我不竭的动力之源。希望他们在走上社会几年后，能够渐渐体会到上我这门课的好处。

目　　录

开篇语　/ 1

任何足够先进的技术都与魔术无异。

——阿瑟·C.克拉克，“技术及未来前景”，

《三号行星的报告》，1972年

第一部分　硬件

计算设备的历史悠久，不过早期的计算设备大多数是专用的，通常用于预测天文事件及其发生方位。例如，关于巨石阵，一个尚未证实的推测就认为它是一座天文观测站。公元前100年制造的安提基瑟拉机器就是一台天文计算机，其机械结构之精妙令人叹为观止。

第 *1* 章　计算机里有什么　/ 11

PC这个名字是个人计算机（Personal Computer）的缩写，或者说源自于1981年IBM开始卖的那种PC。还有些人可能有苹果Mac机，上面运行某个版本的Mac OS X操作系统。更专用的设备，比如手机和平板电脑，也是强大的计算机。这些计算机看起来很不一样，用起来也感觉不一样，但这仅仅是表象，其实根本没区别。

1.1　逻辑构造　/ 13
1.2　物理构造　/ 18
1.3　摩尔定律　/ 21

第 2 章　比特、字节与信息表示　/ 23

温度计的红色液体（通常是染色酒精）或水银柱是模拟的：液体会随着温度变化按比例膨胀或收缩，因此温度产生较小的变化，液体柱高度也会相应产生较小变化。但大楼外面显示温度的广告牌则是数字的：显示屏显示的是数值，温度介于36.5和37.4之间时，它都显示为37。

2.1　模拟与数字　/ 23
2.2　模数转换　/ 25
2.3　比特、字节与二进制　/ 30
2.4　小结　/ 37

第 3 章　深入了解 CPU　/ 39

今天的笔记本电脑，甚至连手机都已经有多个CPU了。英特尔酷睿双核处理器在一块集成电路芯片上集成了两个CPU（“核心”）。在一块芯片上集成越来越多的处理器已经成为明显的趋势。

3.1　玩具计算机　/ 40
3.2　真正的 CPU　/ 45
3.3　缓存　/ 47
3.4　其他计算机　/ 49

硬件部分小结　/ 51

第二部分　软件

2010年4月的美国《消费者报告》（*Consumer Reports*）称丰田雷克萨斯GX460车型“不能买：存在安全隐患”，因为其电子稳定控制系统会导致这款SUV在急速转弯时车尾过分向外甩，从而可能导致翻车事故。一个月之内，丰田公司就升级软件，修复了这个问题。根本就没有任何机械问题，仅仅是软件最初有些问题。

第 4 章　算法　/ 57

假设我们想找出谁是房间里个子最高的人。我们可以四下里看看，然后猜一猜会是谁。然而，算法则必须精确地列出每一个步骤，从而让不会说话的计算机都能遵照执行。最基本的做法就是依次询问每个人的身高，并记住到目前为止谁最高。于是，我们可能会问“约翰，你多高？玛丽，你呢？”

4.1　线性算法　/ 58

4.2　二分搜索　/ 60

4.3　排序　/ 62

4.4　难题与复杂性　/ 66

4.5　小结　/ 68

第 5 章　编程与编程语言　/ 71

程序必须考虑实际的问题，比如内存不足、处理器速度不快、无效或恶意的输入、网络连接中断，以及（看不见摸不着，但却经常会导致其他问题恶化的）人性弱点。因此，如果说算法是理想化的菜谱，那程序就是让烹饪机器人冒着敌人的炮火为军队准备一个月的给养所需的操作说明书。

5.1　汇编语言　/ 72

5.2　高级语言　/ 73

5.3　软件开发　/ 79

5.4　软件资产　/ 84

第 6 章　软件系统　/ 93

你使用的电脑中会装有各种各样的程序，比如浏览器、文字处理器、音乐播放器……这些程序有一个专业的叫法，即应用程序（application）。典出何处？或许出自“这个程序是计算机在完成某个任务方面的应用”吧。

6.1　操作系统　/ 94

6.2　操作系统怎么工作　/ 98

6.3　其他操作系统　/ 101

6.4　文件系统　/ 102

6.5　应用程序　/ 108
6.6　软件分层　/ 111

第 7 章　学习编程　/ 115

如果你自己折腾一天连10行代码都调试不好，那别人要是说能按时交付百万行级的程序，而且没有任何bug，你相信吗？换个角度说，有点编程常识也能让人明白，其实也不是写什么程序都那么难，大不了请人帮你写呗。

7.1　编程语言的基本概念　/ 116
7.2　第一个 JavaScript 程序　/ 117
7.3　第二个 JavaScript 程序　/ 119
7.4　循环　/ 121
7.5　条件　/ 122
7.6　库和接口　/ 124
7.7　JavaScript 怎么工作　/ 125

软件部分小结　/ 127

第三部分　通信

几千年前，人们就曾通过善于长跑的人传递消息。公元前490年，费迪皮迪兹从马拉松战场奔跑了42公里到达雅典，把打败波斯人的胜利消息传递给了雅典人。不幸的是，跑到雅典之后，他上气不接下气地说完“庆祝吧，我们胜利了”之后就死了（至少传说里是这么讲的）。

第 8 章　网络　/ 135

电话网作为一个覆盖全球的大型网络，从一开始只传送语音，到后来同时传输语音和可观的数据，为人类做出了贡献。大约有近20年的时间，人们都是通过电话网把家用计算机接入互联网的。

8.1　电话与调制解调器　/ 136
8.2　有线和 DSL　/ 136
8.3　局域网和以太网　/ 138
8.4　无线网络　/ 141

8.5　手机　/ 144
8.6　小结　/ 147

第 9 章　互联网　/ 149

简单算一下就会发现，IPv4地址只有大约43亿个，甚至还不够地球上每人分一个。因此，按照人类使用的通信服务数量的增长势头，这些IPv4地址迟早会被耗光。实际情况比这种“危言耸听”更糟糕，因为IP地址是按块划分的，这样用起来就没有理论上那么有效率。

9.1　互联网概述　/ 150
9.2　域名和地址　/ 153
9.3　路由　/ 157
9.4　协议　/ 159
9.5　高层协议　/ 162
9.6　带宽　/ 172
9.7　压缩　/ 173
9.8　错误检测和校正　/ 176
9.9　小结　/ 177

第 10 章　万维网　/ 179

万维网的诞生可以追溯到1989年。当时，在日内瓦附近的欧洲核子研究中心工作的英国物理学家蒂姆·伯纳斯–李，为便于通过互联网共享科学文献和研究结果而设计了一套系统,以及一个只能用文本模式查看可用资源的客户端。这套系统在1990年投入使用。说来惭愧，我1992年10月还亲眼见过有人使用它，可当时并没觉得它有那么好，也根本没想到6个月后诞生的第一个图形界面浏览器会改变世界。瞧我这眼光！

10.1　万维网如何工作　/ 180
10.2　HTML　/ 182
10.3　表单　/ 183
10.4　cookie　/ 184
10.5　动态网页　/ 186
10.6　网页之外的动态内容　/ 189
10.7　病毒和蠕虫　/ 190

10.8　万维网安全　/ 192
10.9　密码术　/ 201
10.10　小结　/ 208

第 *11* 章　数据、信息和隐私　/ 211

隐私常常就是安全的同义词。至少对每个个体而言，如果自己的生活信息被传播得随处可见，那怎么会让人感觉安全无忧呢？特别是互联网，它对个人安全已经产生了重大影响。这种影响更多体现在财务风险而非人身安全方面。因为互联网让人们从各种来源收集和整理信息变得异常容易，从而为电子入侵大开方便之门。

11.1　搜索　/ 212
11.2　跟踪　/ 216
11.3　数据库、信息与聚合　/ 221
11.4　隐私失控　/ 224
11.5　云计算　/ 225
11.6　小结　/ 230

第 *12* 章　结束语　/ 231

最后，读者诸君务必牢记一点，无论今天的技术多么千变万化，人是不变的。无论从哪方面来看，现代的人类与几千年前的人类并没有太大区别。

注解　/ 236

词汇表　/ 242

索引　/ 253

版权声明　/ 256

开　篇　语

任何足够先进的技术都与魔术无异。

——阿瑟·C.克拉克，“技术及未来前景”，《三号行星的报告》，1972 年

这里有两个故事，一个是天下大事，另一个则完全是我个人的居家琐事。

2011 年 1 月，发生在突尼斯和埃及的民众革命推翻了执政多年的独裁者，为这些饱受压迫之苦的国家带来了一丝自由的气息。在这两次事件中，无数普通民众通过 Twitter 传播最新动向和集会地点，拿起手机拍摄民众抗议和警察施暴的照片、视频，传到 Facebook 和 YouTube 来激励同胞并告知全世界。在突尼斯，在最紧张的那些日子里，民众发帖量几乎达到了每小时 2 万条，然后被转发到 Facebook 和博客上，又被再次推送。在埃及事件期间，Google 马上开发了一个叫 speak2tweet 的应用程序，这样任何人都可以拨通电话，口述想要发帖的内容，然后由后台系统转换成文本发送出来。人们可以看到这些原始的消息（一般是法语或阿拉伯语），或者让系统朗读出来，或者用 Google 的翻译服务译成别的语言。在这两个国家，尽管革命能否带来长远变化仍然扑朔迷离，但显而易见的是，面对突发事件，政府想控制言论的努力已经是竹篮打水一场空。

2010 年圣诞节的前几天，我想买个更大的电视机，却拿不定主意。于是打开安卓手机，对着搜索引擎说：大屏幕电视。于是返回十多条结果，都是 45 英寸以上的电视机信息。大部分都太贵了，不过其中有一台 650 美元的看起来还不错，于是我点进那个链接，系统提示可以网购或者查看本地实体店的情况，后者则列出了好几个商店，按照

到我家的距离从近到远排好——当时我正坐在家里。顾客给电视机和商店的评分一目了然。我可以马上下单、提货，还能查看从我家到这个实体店的行车指南，包括时间、里程甚至最近路线如何走的说明，在地形图和街区级道路鸟瞰图上，还叠加了当前的交通路况。最后我决定还是先不买了，因为我看电视不多，现在用的这台电视机已经够用了，不过我还是感受到了很大的诱惑。

上面两个故事听起来像魔法不？或者是你已经对这些高科技把戏习以为常了？神奇也好，习惯也罢，这就是今天的技术，甚至算不上先进了，当然更不是魔法。如果一定要说有魔法，那这魔法大概就体现在过去一二十年间技术领域里各种变革的程度之广和速度之快，以及技术进步给人类生活带来的不可思议的影响。

让我们回顾一下当今的一些流行技术在 20 年前也就是 20 世纪 90 年代是什么样子的。如果你现在还是在校大学生或者刚刚毕业，那么当时你还是襁褓之中的小屁孩。那时虽然距离恐龙从地球上消失已经很久了，但我之前讲的事情确实还都不存在。

电话。在 1990 年，电话机还都是傻大个儿，用转盘或者 12 个数字按钮拨号，没有显示面板。它通过电话线连到家里或办公室墙上的插座，只能用来跟人讲话，或者通过特殊设备连接传真机和计算机。到了 2000 年，手机已经流行起来，但也只能打打电话。而现在手机已经非常普及，在几乎所有地方都超过了固定电话装机数量。比如埃及和突尼斯，有手机的人已经分别占总人口数的 80%和 75%之多。智能手机则可以通过从应用市场下载并安装程序来扩展出各种神奇的功能，例如苹果公司在 2007 年中发布了 iPhone 手机，2008 年 App Store 开张，为 iPhone 提供各种软件；安卓市场也类似。

相片。在 1990 年，大部分相机都使用胶卷来拍摄相片。胶卷就是一卷柔软的塑料片，上面覆盖了特殊成分的感光材料。拍摄的时候，光线照射到底片上，化学性质发生改变，留下影像；拍摄完后，需要利用一系列精致的化学反应进行冲洗显影。一卷胶卷最多可以拍摄 36 张照片，冲洗之前并不能看到照出来的是什么样子，并且冲洗过程在任何地方最短也要一个小时，多的长达一星期。这一切都完成之后，你得到的也只不过是一份印好的相纸。如果想和朋友分享，你只能把相片拿给他们看，或者装在信封里寄过去。你的相片可能放在纸袋子或者相册里，散落在家中的各处。若是想再多要一份相片，你就要奔波到照相馆去花额外的钱。那时候摄影开销相当贵，导致大多数人并不会去拍很多相片。但那时数码相机已经露出取代胶片相机的苗头，从慢慢入

侵到迅速占领，直到后来，胶卷从普通人的视线中几近消失。如今，相片已经可以用电子邮件发送给别人或上传到 Facebook、Flickr 等网站，并且可以下载到数码相框来充分展示。如果你还需要纸质相片，那么可以用高品质的相片打印机，它也已经很便宜了。手机摄像头则正在取代廉价的傻瓜相机。

音乐。在 1990 年，音乐是通过 CD 或录音带专辑来发行的，密纹唱片（LP）虽然还很常见，但已经快要过时了。如果你想要复制一首歌和朋友分享，或者放在车里听，那么最常见的办法是使用卡带录音机，尽管也有办法复制 CD 盘片。那时候还没有下载音乐这一说。在 1999 年，Napster 横空出世改变了一切，让大家可以通过网络共享音乐。不过，它很快就关闭了，或者准确点说，被唱片界的传统势力所扼杀。但后来的在线销售单曲服务却把它的遗志发扬光大，比如 2003 年开张的 iTunes 商店就很红火。在便携式硬件方面，我们则经历了便携式收录机、卡带随身听、CD 播放机、以 iPod 为代表的 MP3 播放器，直到后来用手机来听歌。

电子邮件。20 年前用电子邮件的人屈指可数，大多数人甚至都对此一无所知。1998 年，梅格·瑞恩和汤姆·汉克斯主演的《电子情书》才让电子邮件走入公众视野，而之前人们对“邮件”一词的认识仅限于邮递员送来的信件。现如今，大多数人都用 Gmail 之类的在线邮件服务进行个人通信。他们的邮件保存在互联网上，这样用手机也可以访问。

Facebook。1990 年还没有 Facebook，因为这家网站是在 2004 年由年方二十的马克·扎克伯格一手创建的。20 年前如果你要和朋友保持联络，一般要通过电话或者写信，10 年前往往用电子邮件，而如今则有很多人使用 Facebook 发布消息和发送短信。

YouTube。当然，1990 年也没有 YouTube，它比 Facebook 还晚出现一年。如果你要在 1990 年和朋友分享家庭录像，那比分享相片还艰难，因为录像机粗大笨拙，录像带又非常昂贵。而如今，大多数手机已经可以拍摄效果不错的录像，存储卡也价廉而轻便。到 2011 年年中，人们在 YouTube 网站每分钟上传的录像总量可以播放将近 50 小时。

Twitter。它创建得更晚，在 2006 年中期。那时候，博客历经了 7 年的发展已经相当流行，而 Twitter 则首次引入“微博”的创意，让大家可以把内容迅速转发给大量听众。

语音处理。在 1990 年，几乎不可能让计算机理解口语，后来也仅仅是在实验室环境下才能识别有限的词汇。现在，只要给商家打个技术支持或投诉电话，就不可避免要

和语音识别系统进行“交谈”，遇到真人接线员的机会则变得极少。这并不算什么了不起的进步。机器翻译也差不多是同样的情况，虽然尚未达到完美，但已经很实用：它可以把将近 60 种语言中的任意一种翻译成其他语种。语音合成，也就是根据文本发出语音，尽管还很容易听出与真人语音的差别，但也已经很常用了。

购物。20 年前还没有网络商店和网络购物，与之最像的大概是根据西尔斯百货、里昂・比恩户外用品店这些销售商寄送的纸质产品目录进行邮购。如果想比较产品功能，或者寻找最佳售价，你或许会用到邮箱里不请自来的广告传单，但更大的可能是仍然必须亲自去趟商店作比较。产品评测就只能去看《消费者报告》之类的杂志了。到如今，你待在家里就能用手机打开搜索引擎，找到实体店和网店的链接，查看价格和评测，或者在店里对着商品扫一下条码就能查到想要的信息。

地图。在 1990 年，如果你不知道如何到一个地方去，那就要先在电话黄页本上找到地址，打开纸质地图找找看在哪里，然后一边开车一边对着地图人肉导航。迷路之后要自己弄明白身处何地，如何回到正确路线。还要事先猜测交通路况，或者寄希望于交通广播台。如果想知道从天上看道路是什么样子，那只能去包飞机了！10 年前出现的 GPS 导航设备为行车人解决了大部分导航问题，如今手机内部已经集成了具有类似功能的系统。用街景地图就可以在到达一个地方之前先观其大略。

本书的目的就是揭开魔法的神秘帷幕，让读者了解这些系统是如何运作的。相片、音乐和电影如何能一瞬间传遍全球？电子邮件是如何运转的？你的电子邮件私密性如何？为什么垃圾邮件容易发送却难以清除？手机真的知道你的位置吗？iPhone 和安卓手机有什么区别，为什么它们在根本上又是一回事？读过本书之后，你将会对计算机和通信系统的运转有相当靠谱的了解，并知道这些技术如何影响我们的生活。

在本书后面的章节里，我将要为大家讲述一些重要的底层概念。这些概念，贯穿于上面提到的按硬件、软件、通信划分的组织方式中。

首先是**信息的通用数字表示**。传统上用来存储相片、音乐等不同类型信息的机制是错综复杂的，而现在它们已经被一种统一的机制所取代。这种取代之所以可行，是因为信息被表示为数字形式而不是专门形式（比如底片上曝过光的感光材料，或者录音带上的磁化布局）。纸质邮件被数字邮件所取代，纸质地图被电子地图所取代。总之，信息的不同模拟表示形式被统一的数字表示形式所取代。

其次是**通用数字处理器**。所有的信息都用数字计算机这样一种统一的设备进行处理。处理信息通用数字表示的数字计算机代替了处理专门的模拟信息所用的实体机器。行文至此，我一直在竭力避免使用“计算机”这个词，但实际上手机就是相当复杂的计算机，其运算能力已经比得上五六年前的笔记本电脑。我们稍后会发现，不同的计算机在“能计算什么”上的能力是一样的，差别只在于计算速度有多快，能存储的数据有多少。以前只能在台式机和笔记本电脑上做的事，必然会越来越多地可以在手机上完成。如果有什么要特别指出的话，那就是，这个趋同化的过程只会越来越快。

再次是**通用数字网络**。互联网把处理数字表示的数字计算机联接在一起。计算机、手机被联到邮件、搜索、社交网络、购物、网上银行等各种服务上。与别人互发电子邮件的时候，完全不需要考虑对方在哪里、选择用什么方式存取电子邮件。用笔记本电脑、手机、平板电脑都可以搜索商品、比较价格、下单购物。社交网络也让你通过手机和计算机与家人朋友保持联系。显然，要让所有这些计算服务正常工作，网络这个基础设施的影响是巨大的。

最后，海量的**数字化数据**也在持续不断地被收集和更新。全球的地图、航线和街区照片都可以免费获取。搜索引擎为了有效地应对查询而孜孜不倦地扫描着整个互联网。成千上万的书都做成了数字形式。社交网络和资源分享站点为我们保存了关于我们自己的巨量数据。当我们访问网上商店和服务时，它们一方面让我们读取其后台数据，另一方面又在搜索引擎和社交网络的协助和怂恿下默默记录着我们的一举一动。互联网服务供应商记录下我们在网上所有互动操作的联接信息，或许甚至更多。

这一切变化得太快了，数字系统变得越来越小巧，越来越快速，也越来越便宜，而且这种进步呈指数式增长：每过一两年，以同样价格就可以买到性能翻倍的产品，或者说同样的计算能力只需要付出一半钱即可得到。具有更时髦功能、更靓丽屏幕、更有趣应用的新手机不断面世。新的组件程序一直在推出，其中最实用的那些功能则随着时间推移而被直接加入到手机里。这种现象是数字技术天生的副产品：技术发展会导致各种数字设备一刀切式的进步。如果一种技术变化能让数据处理变得更廉价、更快或提升数据处理量，那么所有设备都会从这种变化中得益。结果就是数字系统在我们的生活中无孔不入，成为台前幕后都不可或缺的有机组成。

这种进步，看上去毫无疑问很棒的样子，事实上在大多数情况下也的确很棒，但有时

候也会在人们的心中投下不安的阴影。就个体而言，技术对个人隐私的影响就是其中最显著也是最令人担心的一方面。在我搜索电视机然后转跳到网店的过程中，所有当事方的系统里都留下了我访问和点击的记录。他们如何知道我是谁？哦，因为我用了手机，可以独一无二识地别出我来。他们又如何知道我在哪里从而给我列出附近商店？当然也很简单，手机一直在报告我们的位置。我的手机装有 GPS，于是手机公司可以定位到我家周围 5 米的精度；就算没有 GPS，也可以大致定位到几百米的范围。我们竟然没有经过深思熟虑就允许手机跟踪我们的位置，相比之下，奥威尔在《1984》里所描述的那种严密监视根本就是马马虎虎，表面文章！

其次，对于我们在网上所作所为和往来踪迹的记录看起来会永存不灭。数字存储已经如此廉价，而数据的价值却非常宝贵，于是很少有人会丢弃数据。如果你在网上发表了令人尴尬的不当言论，或者电子邮件发出去后又觉得不妥帖，那么是没有后悔药吃的。从不同来源获得的关于你的信息将让别人有机会拼凑出你生活细节的方方面面，从而被用于商业和政府的用途，而这一切你却一无所知，也未曾授权。这些信息很可能会一直保留着，说不定将来哪一天就会冒出来，让你下不来台。

另外，保护我们隐私和财产的社会机制还没跟上技术的高速发展。20 年前，我去一家本地小银行办业务，我和柜员们互相认识，他们也熟知我的习惯。当发现问题时，他们会为我改正，也能注意到不正常的账户变动。这种亲切的本地服务持续了很久，直到这家小银行被一家全国性的大银行吞并。而在大银行那里，我感觉自己只是账户里的数字而已。银行、我自己或第三方犯下的某个错误，就可能会让千里之外的一个陌生人清空我的账户、盗用我的身份或损毁我的信用记录，天知道还有什么更糟糕的。并且谁也不会马上发现异常。通用网络和通用数字信息让我们面对陌生人时变得脆弱，而这种脆弱程度在一二十年前是不可想象的。同时，在音乐、电影、图书等作品数字化之后，其合法的著作权人在面对未经授权的非法拷贝广泛传播时，也显得无能为力。

本书将帮助你理解这些系统如何运转，以及它们如何改变我们的生活。当然，这只是对当前技术做的一个快照式描述，毫无疑问，五到十年后现在的系统一定会看上去笨重过时。技术变革并不是单个孤立事件，而是一直在进行中的过程：迅速、持续不断而且越来越快。幸好数字系统的基本思想还是不变的，所以如果理解了本书讲解的内容，就能理解未来的系统，这样你就可以更好地面对未来系统所带来的问题和机遇了。

第一部分

硬　　件

硬件是计算机中可见的实体部分，也就是看得见摸得着的设备和装置。计算机设备的发展史相当有趣，但我并不打算在这里讲太多。然而，有些历史趋势却值得我们关注，尤其是这一点：以一定的成本，在给定大小的空间内能装进电路和设备的数量，随着时间而呈指数式增长。随着数字设备越来越强大和廉价，林林总总的机械系统已经被更为统一的电子系统所代替。

计算设备的历史悠久，不过早期的计算设备大多数是专用的，通常用于预测天文事件及其发生方位。例如，关于巨石阵，一个尚未证实的推测就认为它是一座天文观测站。公元前 100 年制造的安提基瑟拉机器就是一台天文计算机，其机械结构之精妙令人叹为观止。算盘之类的演算工具也已经使用了近千年时间，在亚洲尤为流行。计算尺发明于 17 世纪早期，也就是约翰·纳皮尔提出对数概念没多久之后。在 20 世纪 60 年代读大学时，我还用过计算尺，但现在这玩意已经成了古玩。我当初辛苦掌握的计算尺技艺如今也没了用武之地。

与当今的计算机最沾边的前辈是雅卡尔织布机，由法国的约瑟夫·玛丽·雅卡尔大约在 1800 年发明。雅卡尔织布机用长方形卡片上打穿的多行孔洞来标记织布的花样。这样，我们就可以在穿孔卡片上编入各种指令来给雅卡尔织布机“编程”，控制它织出各种不同花样的布来。

虽然尚存在争议，但一般认为，当今意义上的计算机始于 19 世纪的英国，由查尔斯·巴

贝奇提出。巴贝奇是一位科学家，对航海和天文学感兴趣，而这两项事业都需要通过写满了数值的表格来计算方位。巴贝奇花费了毕生精力来制造用于计算的设备，试图把冗长乏味、易出错的手工算术运算机械化。但由于各种原因，包括与资助人之间的关系疏离，他的雄心壮志始终没有得偿所愿。不过，他的设计是正确的，现代人利用他那个时代的工具和材料按其设计可以制造出他的机器。如今，在伦敦的科学博物馆、加州山景城的计算机历史博物馆等地，都能看到这样的机器。

一位年轻的女士受巴贝奇鼓舞而对数学和他那个计算设备产生了兴趣。这位女士就是诗人乔治・拜伦的女儿，奥古斯塔・爱达・拜伦，也就是后来的勒芙蕾丝伯爵夫人。她写过一份详细说明，讲述如何用分析引擎（巴贝奇所计划制造机器里最高级的一个）进行科学计算，并推测这种机器也可用于非数值计算，比如作曲。爱达・勒芙蕾丝通常被认为是世界上第一位程序员，编程语言 Ada 也是以她的名字而命名的。

在 19 世纪后期，赫尔曼・何乐礼为美国人口统计局设计并制造了制表机，用它制作人口统计数据表格要比手工快得多。何乐礼借用了雅卡尔织布机的思路，用卡片纸上的孔洞把人口统计数据编码成他的机器能处理的格式。令何乐礼名声大噪的是，1880 年的人口数据花了六年才制成表，而用了他的穿孔卡片和制表机之后，1890 年的数据仅一年就完工。他创立了一家公司，经过多次并购之后成为国际商业机器公司，也就是我们现在熟知的 IBM。

巴贝奇的机器是由齿轮、转轮、杠杆、拉杆组合起来的复杂机械，而 20 世纪电子学的发展使得人们有条件去设想没有运动部件的计算机会是什么样子。到了 20 世纪 40 年代，在费城的宾夕法尼亚大学，由布莱斯波・埃克特和约翰・莫奇利设计制造的 ENIAC（电子数值积分计算机的英文首字母缩写）横空出世，成为全电子计算机的最重要标志。ENIAC 占满了一间大屋，需要消耗很多电力，每秒钟能做大约五千次加法。它本来是为了计算弹道等军事用途而制造的，但直到 1946 年"二战"已结束多时它才完工。ENIAC 的一些部件，现存放在宾大的摩尔工程学院作展览。

巴贝奇清楚地意识到，计算设备可以把操作指令和数据保存为同样的形式，但 ENIAC 并没有把指令像数据那样保存在存储器中，而是通过扳动开关和重新连线来实现编程。第一台真正实现了存储程序的计算机于 1949 年在英国面世，称为 EDSAC（延迟存储电子自动计算机的英文首字母缩写）。

早期的电子计算机使用电子管作为基本计算元件。电子管是一种大小和形状类似于柱形电灯泡的电子设备，缺点是昂贵、易碎、笨重、费电。而随着 1947 年晶体管和 1958 年集成电路的相继发明，计算机的新时代才真正开始。用此技术制造的设备是当今的电子系统越来越小、越来越迅速以及越来越便宜的原因所在。

第 1 章

计算机里有什么

让我们开始讨论硬件，先大略看看计算机里面都有些什么东西。这个问题可以从两方面来看：逻辑上或者说功能上的组成，即每一部分是什么、做什么、怎样做、之间如何连接；以及物理上的结构，即每一部分长什么样子、如何建造起来的。本章的目标是了解计算机里有什么，大致弄清每一部分做什么，并能识别出各种首字母缩写和数字的含义。

回想一下你自己的计算机。许多读者可能会有 PC 机，上面运行着来自于微软公司的某个版本的 Windows 操作系统。PC 这个名字是个人计算机（Personal Computer）的缩写，或者说源自于 1981 年 IBM 开始卖的那种 PC。还有些人可能有苹果 Mac 机，上面运行某个版本的 Mac OS X 操作系统。更专用的设备，比如手机和平板电脑，也是强大的计算机。这些计算机看起来很不一样，用起来也感觉不一样，但这仅仅是表象，其实根本没区别。为什么这么说呢?

可以拿汽车来打个大致的比方。在功能构成上，这一百多年来的汽车都是一样的。每辆汽车都有个发动机，通过燃烧某种燃料来驱动发动机运转，这样车就能开了；都有个方向盘，这样司机（也许可以比作软件）就能操纵车的方向；还有储存燃料的地方，以及留给乘客和行李的位置。但是这一个多世纪以来汽车在物理构成上却变化巨大：造车的材料日新月异，行驶越来越快，越来越安全，可靠性和舒适性与过去不可同日而语。我的第一辆车是一部饱经风霜的 1959 款大众甲壳虫，跟法拉利比起来有天壤之别，但不论是载着我和买来的杂货从附近商店回家，还是穿越整个国家，它们都能各

尽其职。就这个意义来说，它们在功能方面是一样的。（不过要申明一点，我没坐过法拉利，更不要说自己拥有一辆了，所以我只是在猜测它有没有放杂货的地方。）

计算机也是一样的道理。当今的计算机在逻辑结构上和 20 世纪 50 年代的非常相似，但是物理指标的进步却远甚于汽车。当今的计算机和 50 年前的比起来，体积更小，价格更廉，运行更快，也更可靠，有些字面上的指标甚至提高了百万倍。计算机如此普及，其根本原因就在于此。

一件东西的功能表现与物理特性之间的区别，也就是说它能做什么与它是怎样建造起来的（或者说内部的工作方式）之间的区分，是很重要的。就计算机而言，"它是如何建造出来的"这个问题的答案在以惊人的速度变化着，"它运行起来有多快"也是如此，但是"它能做什么"的答案却没什么变化。后面将会反复提到这两方面的区别。

有时我会在第一堂课时在班级里作个调查，看多少人有 PC 机，多少人有 Mac 机。在新千年的头几年，PC 对 Mac 一直稳定在 10 对 1 的比例，但几年后形势突变，到现在 Mac 已经占了全班同学机器的四分之三还要多。尽管就全世界范围而言仍然是 PC 占统治地位，我班里的这种变化肯定不是典型情况，但变化确实是存在的。

那么，两者的比例严重失调是因为一种机型比另一种更好吗？如果是的话，那又是在什么方面发生了惊人的变化，才导致在这么短时间里就形势大变呢？我询问过学生他们认为哪种机型更好，以及他们是根据什么客观标准作出这种判断的。在购买计算机的时候，你是根据什么选择了现在拥有的机型呢？

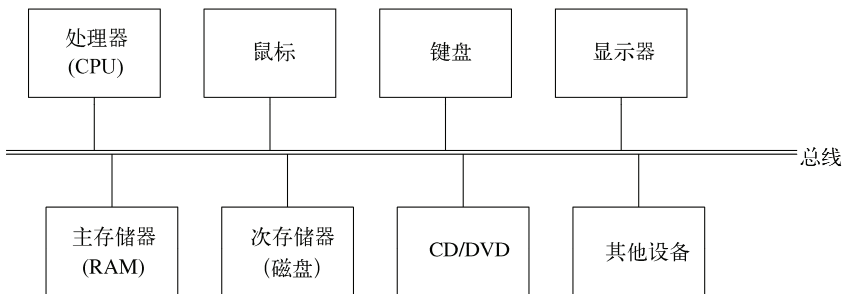
价格当然是一个原因。PC 的价格更便宜些，这是多家供应商在市场上激烈竞争的结果。有各种各样的 PC 硬件外设、数量众多的 PC 软件，关于 PC 的各种专业知识也更容易获取。（这可以看作是经济学里网络效应的实例：使用某种东西的人越多，对其中每个人来说这种东西也就越有用，两者之间基本成正比关系。）

而对选择 Mac 的人来说，他们自认为 Mac 在可靠性、质量和美学设计上更胜一筹，还有人强调"到手就能用"，甚至还包含几分对 PC 的恐惧和厌恶。这些因素让他们愿意付出更多的钱。

争论一直在进行，谁也说服不了谁。但是由此引发了一些有价值的问题，让人们思考这两种计算机的差别在哪里，共同之处又是什么。

1.1 逻辑构造

如果我们画一张抽象图展示计算机内部有什么，也就是它逻辑上或者功能上的体系结构，那么 Mac 和 PC 的结构都是如下图所示：一个处理器（CPU）、一些主存储器（内存）、一些大容量存储器（磁盘）和各种各样的其他部件，一组叫做总线的线缆把所有这些连接起来，在各部件之间传输信息。



计算机的基本组成，包括处理器、存放指令和数据的存储器以及输入输出设备，在 60 多年前就已经是标准了。这种体系结构通常称为冯・诺依曼体系结构，以约翰・冯・诺依曼的名字命名。他在 1946 年与阿瑟・勃克斯、赫尔曼・戈德斯坦共同撰写的论文《电子计算仪器逻辑设计的初步讨论》中描述了这种体系结构。尽管目前人们对于以冯・诺依曼来命名这种体系结构是否掩盖了其他人的贡献尚有争议，但这篇论文条理清晰，见解深刻，即便在今天也值得一读。例如，论文的第一句就指出："为了让这台完整的设备成为通用的计算机器，它必须包含某些主要元件用于运算、存储数据、控制以及连接操作人员。"翻译成现在的术语就是，CPU 提供运算和控制功能，内存和磁盘用于存储数据，键盘、鼠标和显示器用于连接操作人员。

1.1.1 CPU

如果我们说计算机有大脑的话，处理器，或者叫中央处理单元（缩写为 CPU）就是计算机的大脑。处理器进行运算，来回搬运数据，并控制着一切别的操作。CPU 有一张指令表，它可以执行的操作是有限的，但执行起来速度异常之快，高达每秒钟几十亿

次。它可以根据先前的计算结果决定接下来执行什么指令，所以在很大程度上，它可以主宰自己的命运。这一点很重要，我们将在第 3 章花更多篇幅详述。

如果你去商店或者上网购买计算机，你将会发现上面提到的大多数部件都会跟着一串神秘的缩写和同样神秘的数字。比如你可能看到对 CPU 的描述是“英特尔双核酷睿 2.1 GHz”。这是什么意思呢？这款 CPU 是英特尔制造的，一片封装的内部实际上有两个 CPU。在这句话里，“核”的意思就是处理器。

2.1 GHz 看起来更有趣。CPU 的速度大体上是以每秒钟执行的操作数量、指令数量或更小的动作数量来度量的。CPU 使用一个跟心跳或者钟表嘀嗒类似的内部时钟来控制基本操作的节拍，度量 CPU 速度的指标之一就是看这个内部时钟每秒振动多少次。每秒钟心跳一次或者嘀嗒一次就是 1 赫兹，记为 1 Hz。这个单位名称是为了纪念德国工程师海因里希 · 赫兹，他在 1888 年发现了产生电磁辐射的方法，由此直接导致无线电广播和其他无线系统的诞生。广播电台发射的广播信号频率为兆赫（百万赫兹），比如 102.3 MHz。现在的计算机通常运行在十亿赫兹的数量级上，也就是吉赫，记为 GHz。我的笔记本电脑有一片很普通的 2.1 GHz 处理器，它每秒钟跳动 21 亿次。人类的心跳大约是 1 赫兹，也就是每天大约跳 10 万次，每年将近 3 千万次，于是我的心脏要花 70 年才能赶上这个 CPU 在一秒钟里跳动的次数。

这是我们第一次遇到像兆、吉这样的数字前缀，这些前缀在计算中普遍使用。兆是 100 万，或者说 10 的 6 次方；吉是 10 亿，或者说 10 的 9 次方。我们马上就会遇到更多的计量单位，在书后的术语表里则有一份完整列表。

1.1.2 RAM

主存储器，也就是随机访问存储器（缩写为 RAM，即内存），里面保存了处理器和计算机的其他部件正在活跃使用的信息；CPU 可以改变内存里的内容。内存里不仅保存了 CPU 正在处理的数据，还保存了让 CPU 如何处理数据所需运行的指令。这一点至关重要：通过把不同的指令加载进内存，就可以让计算机做不同的计算。这样，存储程序型计算机就成为通用的设备：同一台计算机，只要在内存里放上适当的指令，就可以运行文字处理程序、制作数据表格、上网浏览、收发电子邮件、计算纳税款，还可以播放电影。存储程序这个理念的重要性怎么强调都不为过。

内存是计算机运行的时候存储信息的地方。运行中的程序，比如 Word、iTunes 或浏览器，它们的指令就放在内存里；这些程序操作的数据，比如屏幕上显示的照片、正在编辑的文档、正在播放的音乐等，也是放在内存里的；而 Windows、Mac OS X 或其他操作系统，也就是能让你在同一时间运行多个应用程序的幕后功臣，它们运行时的指令还是放在内存里。到第 6 章的时候我们会讲到操作系统。

内存之所以被称为“随机访问”，是因为 CPU 能以同样的速度快速访问其中任何地方的信息。以任何顺序随机访问不同位置时，速度不会受到任何影响。与之相比，老式录像带的访问方式则称为“顺序访问”，比如想观看电影尾声的时候只能从头开始慢悠悠地“快进”，跳过前面的内容。

内存是易失性的，也就是掉电之后里面的内容会消失，当前活跃的信息就都丢掉了。所以要养成小心谨慎的好习惯，经常保存正在做的工作，尤其是在台式机上，不小心踢掉电源线可不是闹着玩的。

你计算机上的内存大小是有限的。表示容量的单位是字节。一字节大小的内存，可以放入单个字符（比如 W 或者@），可以放入一个整数比如 42，还可以放入大数值的一部分。第 2 章会展示内存或者计算机其他部件中的信息是如何表示的，因为这是计算的一个基础问题，但在此之前，你可以把内存想象成一大堆完全一样的小盒子，上面从 1 开始编号到一二十亿，每个盒子里可以放进一小片信息。

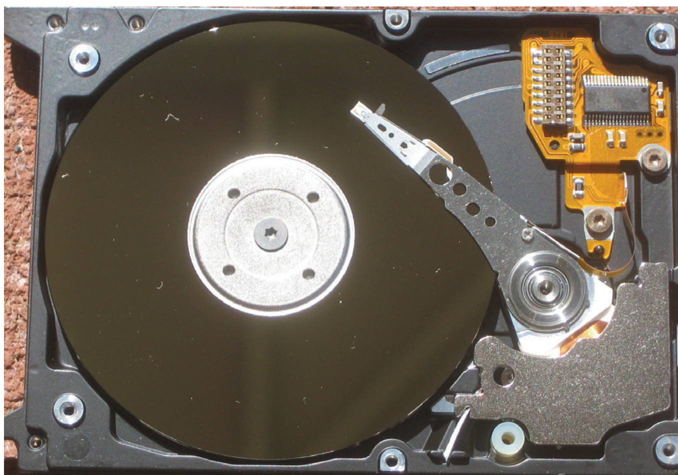
内存的容量有多大？我正在用的这台笔记本电脑，内存有 20 亿个字节，或者说 2 吉字节，也就是 2 GB。有很多人还认为这么些内存太小了点。原因是，虽然所有程序同时开起来的时候，内存再多也不够用，但是内存越大，可供发挥的空间就越大，而这往往也可以说成是计算得越快。如果你想要计算机运行更快的话，多买内存看起来是最佳策略。

1.1.3 磁盘和大容量存储器

内存的容量很大，但还是有限的，并且掉电之后内容会消失。大容量存储器则能在掉电后仍保存着里面的信息。最常见的大容量存储器是磁盘，有时也称为硬盘或硬驱。磁盘能保存的信息比内存大得多，并且是非易失性的，也就是说，磁盘上的信息不论

通电还是断电都一直在那里。于是数据、指令和其他信息都长期保存在磁盘上，仅在需要时临时读入内存。磁盘空间比内存便宜 100 倍，只是访问起来要慢得多。

磁盘保存信息的方法是对旋转的金属盘片表面的磁性材料上的微小区间进行不同方向的磁化。计算机工作时的嗡嗡声和咔嗒声就是磁盘把磁头移向盘片表面正确位置时发出来的。你可以在下图里看到标准笔记本电脑硬盘的盘片表面和磁头。盘片的直径是 2.5 英寸（6.25 厘米）。



有些计算机配备了固态盘（SSD），使用闪存代替了旋转的机械部件。闪存也是非易失性的，信息保存为电路里的电荷，每一个电路元件上的电荷不需要加电就可以保持其状态。里面保存的电荷可以通过测试来读取其中的值，也可以擦掉后用新的值覆盖写入。目前闪存仍然比同容量的传统硬盘贵，但是速度快、轻便、可靠并且节电，在手机、相机等设备中广泛使用，将来可能会取代机械硬盘。

当今一块典型的笔记本电脑硬盘能保存几百吉字节的信息，而插在 USB 插座上的外部硬盘容量则达到了太字节（TB）的数量级。（“太”这个前缀表示 1 万亿，或者说 10 的 12 次方。随着硬盘容量的增长，我们以后会越来越多地见到这个单位。）

1 太字节有多大呢？或者退而求其次，1 吉字节有多大呢？按照最常用的表示文字的方法，1 字节可以保存字母文字中的一个字符。《傲慢与偏见》英文原版印在纸上有大约 250 页，包含大约 55 万个字符，所以在 1 GB 空间里能保存将近 2000 份副本。更

可能的情况是，我只保存一份副本，然后再存点儿音乐。MP3 音乐每分钟大约占 1 MB，那么从我最喜爱的那些 CD 里挑出一张《简·奥斯汀的歌本》转成 MP3 保存进去，大约占了 60 MB，剩下的空间还可以再放进长达 15 小时的音乐。1995 年，BBC 出品了詹妮弗·艾莉和柯林·费斯主演的《傲慢与偏见》电影，DVD 版是两张碟片，容量不到 10 GB，可以在 1 TB 的硬盘里放 100 多次。

硬盘是用来展示逻辑结构和物理实现之区别的好例子。在 Windows 下运行资源管理器或者在 Mac OS X 下运行 Finder，可以看到硬盘里的内容组织为层次分明的文件夹和文件，但真正的数据是完全存放在旋转的机械装置、没有活动部件的集成电路或者其他存储设备里的。计算机里装的究竟是哪种"磁盘"其实无关紧要，事实上是硬盘里的硬件电路和操作系统里称为文件系统的重要软件一起创建了这种有组织的结构。在第 6 章我们会详谈这个问题。

这种逻辑结构跟人的思维相当匹配，或者更合适的说法是，到如今我们已经完全习惯了这种组织方式，所以别的存储设备也提供了同样的组织方式，哪怕是它们使用了完全不同的物理方法来实现存储。比如说，CD-ROM 或者 DVD 使用了看起来跟硬盘上的文件系统一样的方式来储存信息；USB 设备、数码相机和可插存储卡等其他小玩意也都这样；就连现在已经完全淘汰的老古董软盘，在逻辑层次上看起来也完全一样。

1.1.4 其他设备

各种各样其他设备也都发挥着特别的作用。有的让用户提供输入，比如鼠标、键盘和触摸屏；有的为用户提供输出，比如显示器、打印机、扬声器；网络部件，比如以太网或者无线网则用来和别的计算机通信。

在体系结构示意图里，这些设备看上去是通过一组线缆连接在一起的。借用电气工程的术语，这组线缆称为总线。实际上，计算机内部有好几组总线，每组总线都具有适合其功能的特性。比如 CPU 和内存之间的总线，线路短，传输快，但是价格贵；而连接到耳机插孔的总线，线路长，传输慢，但是价格便宜。有些总线在机箱外也露出了一部分，比如无所不在的通用串行总线，也就是把外设插入计算机所用的 USB 总线。

稍后我会在特定的语境下再次提及这些其他设备，现在我不会在这上面花太多时间。我们来列一下连接在计算机上或者从属于它的设备：鼠标、键盘、触摸板和触摸屏、显示器、打印机、扫描仪、游戏操纵器、音乐播放器、耳机、扬声器、话筒、相机、电话、与其他计算机的连接……这个列表还可以继续扩充。所有这些设备，跟处理器、内存和硬盘走着同样的进化路线：它们的物理属性一直朝着性能更强、规格更小、价钱更低这几个方向突飞猛进。

这些设备中，有的正在往一体化的方向融合，这种融合的进展非常值得关注。现在的手机也能当作手表、照相机和录像机、音乐和电影播放器、游戏主机、条码阅读器和导航仪来用，以至于拿手机来跟人通话倒成了偶尔才使用的功能。手机其实和笔记本有着同样的体系结构，只是由于尺寸和电源等约束而表现出较大区别。手机没有硬盘，但是有非易失性的闪存，这样它就能够在关机时仍保存电话本、应用程序和其他信息。手机能连接的外部设备没那么多，但还是可能有蓝牙、耳机和外部话筒插孔和 USB 接口等。

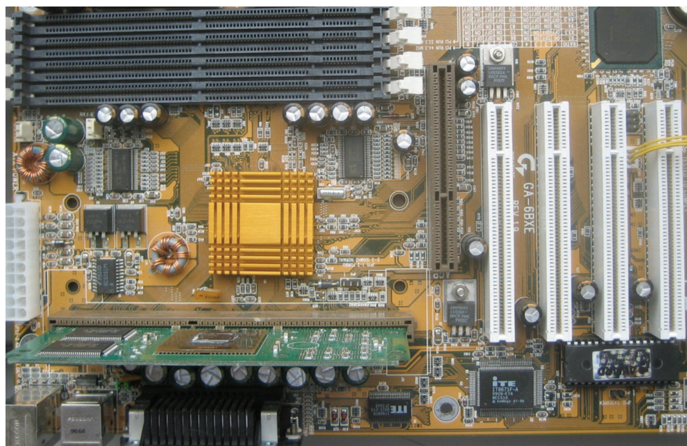
在我写这本书的时候，平板电脑的时代正在开启。像 iPad 以及它的各种竞争对手，很可能会占据另一片重要的天空。这些玩意也是计算机，也有一样的体系结构和相似的部件。

1.2　物理构造

我把自己翻箱倒柜找出来的这几十年间的古董硬件设备拆开拿到班上，传给学生，让他们观察计算机的五脏六腑。由于计算机里面的很多东西都抽象成了逻辑结构，所以能实际观察和触摸硬盘、集成电路芯片、制造芯片所用的晶圆等东西对于学习计算机是很有用的，而观察一些设备的进化史也很有趣。比如现在的笔记本电脑硬盘跟 10 年前的没什么区别，只是容量增大了 10 倍或者 100 倍，但从外表根本看不出来。

另一方面，承载计算机部件的电路板却能看出明显的发展。部件的数量在减少，因为更多的电路被做到了内部，布线更加精细，电路的引脚比起 20 年前多了很多，

也密集了很多。下图展示了一块 20 世纪 90 年代后期的台式机电路板，CPU、内存等部件安装或者插入到电路板上，通过反面的印刷线路连接起来（图上正面看不到）。

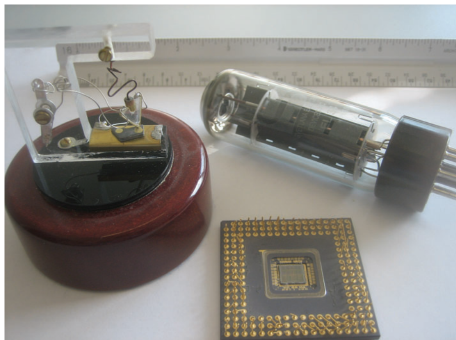


计算机里的电子线路是由大量基本元件搭建起来的，但基本元件的类型却只有很少几种。其中最重要的一种是逻辑门电路，用来根据一个或两个输入值计算一个输出值，也就是用输入的电压或电流信号来控制输出的电压或电流信号。只要把足够多的门电路用正确的方式连接起来，就能执行任何计算。查尔斯・佩措尔德的《编码》是介绍这方面知识的好书，还有一些网站用图形动画的方式演示逻辑电路如何进行数学运算和其他计算。

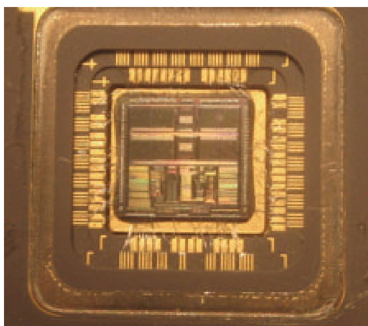
当今最重要的电路元件是晶体管，是 1947 年由约翰・巴丁、瓦尔特・布拉顿和威廉・肖克利在贝尔实验室发明的，他们因此获得了 1956 年的诺贝尔物理学奖。在计算机里面，晶体管基本上就是个开关，也就是用电压控制电流通断的设备。任何复杂系统都可以构建在这么简单的基础之上。

门电路过去是用分立元件搭建的。制造 ENIAC 的时候，用的是跟灯泡差不多大小的电子管，而 20 世纪 60 年代的计算机则用的是铅笔上橡皮头那么大的单独的晶体管。下图展示了第一颗晶体管的仿制品（左边）、电子管和封装起来的 CPU 芯片。电子管大约 10 厘米长，CPU 实际的电路部分是在中间，大约 1 平方厘米。图上这么大的现

代 CPU 芯片里则可以集成上亿颗晶体管。



如今的逻辑门电路是创建在集成电路上的。集成电路（integrated circuits）缩写为 IC，通常也称为芯片或微芯片。集成电路在一个平面里（很薄的硅片）包含了电路板的所有部件和布线，通过一系列复杂的光学和化学流程制造出来，这样就得到了没有分立元件也没有传统布线的电路。因此，集成电路比分立部件的电路小得多也可靠得多。芯片是在直径 12 英寸（30 厘米）的晶圆上批量制造的，然后把晶圆切割成独立的芯片，单独包装。芯片通常安装在比它大很多的封装壳上，用几十到几百条引脚连接到系统的其他部分。下图展示了一片封装起来的集成电路。实际的电路部分在中心，大约 1 平方厘米。



集成电路是用硅制造的，由于这个原因，集成电路产业的发源地加州旧金山南部就有了个外号叫硅谷。现在，这名字已经成为那一带所有高科技产业的统称，并且引起了几十个其他地方的效仿，比如纽约有硅巷、英国剑桥有硅沼等。

集成电路是 1958 年由罗伯特·诺伊斯和杰克·基尔比各自独立发明的。诺伊斯在 1990 年逝世，基尔比则在 2000 年因此获得了诺贝尔物理学奖。集成电路是数字电子设备的主角，但其他技术也发挥了作用，包括硬盘中的磁存储、CD 和 DVD 中的激光、光纤网络中的激光等。过去 50 年里，所有这一切都在尺寸、容量和价格方面发生了惊人的改进。

1.3 摩尔定律

1965 年的时候，戈登·摩尔，也就是后来的 Intel 公司联合创始人及长期 CEO，发表了一篇文章《在集成电路里填入更多部件》。他注意到，随着技术的发展，在给定大小的集成电路内部可以制造的设备（主要是晶体管）数量大约每年翻一番。后来他又把这个速率修正为两年，另外有些人则认为是 18 个月。由于计算能力大体上可以用晶体管数量来代表，这就意味着计算能力只要两年或更短时间就能翻倍，也就是说，20 年下来可以翻十番，集成度提高 2 的 10 次方也就是大约 1000 倍，经过 40 年则可以提高 100 万倍或更多。

这种指数式增长，也就是通常说的摩尔定律，已经持续了 50 年，于是当今集成电路里的晶体管数量早已超过了 1965 年那时候的 100 万倍。巨大的量变引发了质变。摩尔定律的实际图表，尤其是处理器芯片的数据，显示出晶体管数量从 20 世纪 70 年代早期 Intel 8008 CPU 的几千个晶体管，已经发展到现在廉价家用笔记本电脑的处理器的上亿个。

用来描述电路规模的基本数值是集成电路里的特征尺寸（即其中的最小尺寸），比如导线的宽度。在过去的很多年里，这个数值在稳步缩减。我在 1980 年设计的第一片集成电路（也是我设计的唯一一片）用的是 3.5 微米的特征尺寸；如今的很多集成电路，特征尺寸是 32 纳米，也就是 32 米的 10 亿分之一，下一步将会是 22 纳米（“纳”表示 10 亿分之一，或者说 10 的−9 次方，纳米缩写为 nm）。对比一下，一张纸的厚度或者一根头发的粗细是 100 微米，即十分之一毫米。（“微”是百万分之一，或者说 10 的−6 次方，“毫”是千分之一，或者说 10 的−3 次方。）

集成电路的设计和制造是相当复杂的业务，竞争异常激烈。而且制造运行（“生产线”）也很昂贵，新建的工厂可以轻而易举花掉几十亿美元。如果一个公司的技术和资金跟不上，就会在竞争中严重处于劣势。如果一个国家没有这样的资源，就要为了这些技术依赖外国，存在严重的战略问题。

到某个阶段，摩尔定律会失效。以前曾多次有人断定摩尔定律的极限已到来，但后来又发现了突破极限的方法。然而现在，我们已经到了这样的阶段：有的电路里仅包含极少数原子，这么小的结构已经很难控制。CPU 速度已经不再每两年翻一番（部分原因是芯片越快散热越多），但是内存容量仍然在按这个规律翻倍。与此同时，现在的处理器在一片芯片里可以有多颗 CPU。

比较一下今天的个人电脑和 1981 年最初的 IBM PC，对比是惊人的：第一代 PC 的处理器主频是 4.77 MHz，现在一片 2.3 GHz CPU 的时钟频率快了大约 500 倍；第一代 PC 有 64 千字节内存（“千”缩写为 K），现在一台 4 GB 内存的计算机大约是它的 6 万倍；第一代 PC 至多有 750 KB 软盘，没有硬盘，现在机器的磁盘空间则增加了 100 万倍；第一代 PC 的显示器是 11 英寸的，只能在黑色背景上显示 24 行 80 列的绿色字符，而我写这本书的时候所用的 24 英寸显示器能显示 1600 万颜色；在 1981 年，买一台配备了 64 KB 内存、160 KB 单软驱的 PC 要花 3000 美元，相当于 30 年后的 5000 ~ 10 000 美元，而现在买一台 2 GHz 处理器、4 GB 内存、400 GB 硬盘的笔记本电脑只要花几百美元。

20 世纪计算机科学的伟大发现之一是，现在的数字计算机、最初的 PC 以及再往前体积更大、计算能力更弱的老式计算机器，它们在逻辑或者功能上的特性是完全一样的。如果我们不考虑速度、存储容量这些因素，这些计算机可以做完全一样的计算。在第 3 章，我们将进一步探讨这个问题。

第 2 章

比特、字节与信息表示

这一章，我们来讨论计算机表示信息的三个基本思想。

首先，**计算机是数字处理器**。它们存储和处理离散的信息，这些信息表现为不连续的块，具有不连续的值，基本上就是一个个数值。而与之相对的模拟信息，则是平滑变化的值。

其次，**计算机用比特表示信息**。比特就是二进制数字，即一个非 0 即 1 的值。计算机中的一切都用比特来表示。计算机内部使用二进制，而不是人们所熟悉的十进制。

再次，**较大的信息以比特组表示**。数值、字母、单词、姓名、声音、照片、电影，以及处理这些信息的程序所包含的指令，都是用比特组来表示的。

本章关于数字的讨论并不是非看不可，但其背后的思想却非常重要。

2.1　模拟与数字

首先，我们谈一谈模拟与数字的区别。“模拟”（analog）与“类似的”（analogous）词根相同，表达的意思是：值随着其他因素变化而平滑变化。现实生活中的很多事物都具有模拟性质，比如水龙头或汽车方向盘。如果你想让车转个小弯，轻轻打一打方向盘即可，打多打少由你自己来定。拿它跟转向灯作个比较，后者要么开要么关，没有中间状态。在模拟装置中，某些事物（汽车转弯幅度）会随另一些事物（方向盘转动

幅度）的变化平滑而连续地变化。变化过程没有间断，一个事物的微小变化就意味着另一个事物的微小变化。

数字系统处理的是离散值：可能的取值是有限的（转向灯只可能是关闭的或在左右方向打开）。某个事物小小的变化，要么不引发其他事物变化，要么就引发其他事物的突变，使其从一个离散的值跳到另一个离散的值。

比如手表。“模拟”手表有时针、分针和秒针，秒针每分钟转一圈。虽然现代的手表都由内部的数字电路控制，但时针和分针仍然随着时间流逝而平滑移动，而且三根表针都能走遍所有可能的位置。数字手表或手机时钟显示的时间只有数值。显示屏每秒变化一次，每分钟更新一次分钟的值，但不会显示分钟的小数位。

再比如汽车的速度表。大多数汽车都有模拟速度表，速度指针平滑地上下移动，按比例指示汽车的速度。从一个速度到另一个速度的过渡是平滑的，没有间断。与之相对，汽车的 GPS 导航仪则用数字来显示最接近的时速，不管单位是英里/小时还是公里/小时。稍微加点速，速度值会从 65 变成 66，再减点速就又变回 65，但永远不会出现 65.5。

又比如温度计。温度计的红色液体（通常是染色酒精）或水银柱是模拟的：液体会随着温度变化按比例膨胀或收缩，因此温度产生较小的变化，液体柱高度也会相应产生较小变化。但大楼外面显示温度的广告牌则是数字的：显示屏显示的是数值，温度介于 36.5 和 37.4 之间时，它都显示为 37。

这可能会导致一些奇怪的情况。几年前，我在美国高速公路上开车，收听到加拿大电台的节目。加拿大采用公制单位，播音员出于好意，想照顾到所有美加的听众，就如此宣布说：“刚才的一小时，华氏度上升了一度，摄氏度没有变化。”

有人要问，为什么用数字而不用模拟呢？我们这个世界可是模拟的呀，而且手表、速度表等等模拟设备也更容易让人一目了然。但不管怎样，很多现代的技术都是数字的，而且我们这本书也是在讲述数字的故事。外部世界的数据——声音、图片、运动、温度，等等一切，在输入端都会尽可能早地转换为数字形式，而在输出端则会尽可能晚地转换回模拟形式。原因就在于数字化的数据容易处理，无论最初来源是什么，数字化数据都可以用多种方式来存储、传输和处理，但模拟信息则不行。第 9 章将会介绍，通过删除冗余和不重要的信息，还可以压缩数字化信息。为了安全和隐私可以对它进

行加密，可以将它与其他数据合并，可以复制它而不出错，可以通过互联网把它发送到任何地方，可以将它保存到几乎无限种设备中。而对于模拟信息，上述很多做法是根本行不通的。

与模拟系统相比，数字系统还有另一个优势，就是它更容易扩展。比如说，给模拟天文馆增加一颗新发现的星星，专业人员必须辛苦地做出光照效果来；而在数字天文馆，只要在数据文件里添加一行信息即可。我的数字手表可以连续不断地以百分之一秒显示时间流逝，而要让模拟手表做到这一点可就太难了。不过，模拟系统有时候也有它的优势，像泥版、石雕、羊皮纸、图书和照片等古老的媒体，都经历了数字格式未曾经历过的时间考验。

2.2 模数转换

怎么把模拟信息转换为数字形式？我们还是举几个简单的例子吧，先说照片和音乐，通过它们可以说明一些重要的思想。

把照片转换为数字形式，应该是最容易想象的了。假设我们给自家的小猫拍张照片：



胶卷相机的成像，是通过把胶片感光区曝露给被拍物体反射的光线实现的，胶片上不同区域接收到的不同颜色的光量不同，从而影响胶片上的染料。在胶片显影、印相时，彩色染料数量决定了显示出来的颜色变化。

对数码相机来说，镜头把影像聚焦到一块位于红、绿、蓝滤镜后面的矩形感光器阵列上，感光器由微小的光敏探测器组成。每个探测器存储一定数量的电荷，与落在它上面的光量成正比。这些电荷被转换为数字值，照片的数字表示就是这些表现光强度的数值序列。探测器越小，数量越多，电荷测量的结果就越精细，数字化图像就能越精确地反映原始的影像。

传感器阵列的每个单元都由一组能够捕获红、绿、蓝光的探测器构成，每个单元对应一个像素，即像元。3000×2000 像素的图像，包含 600 万个像元，或 600 万像素，对今天的数码相机而言并不算大。像素的颜色通常由三个值表示，分别代表红、绿、蓝光的强度，因此 600 万像素的图像总共要存储 1800 万个颜色值。屏幕在显示图像时，使用的是红、绿、蓝光三元组的阵列，其亮度与像素亮度一致。如果你用放大镜仔细观察手机或电脑屏幕，很容易看到每个独立的彩色块。

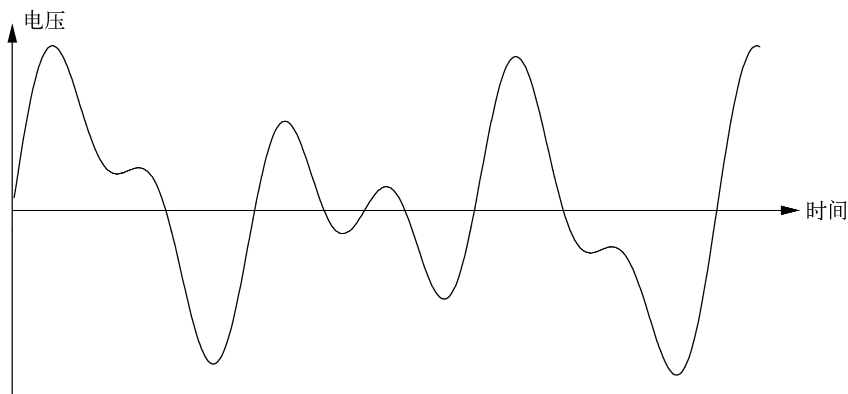
第二个模数转换的例子是声音，尤其是音乐。之所以说音乐是个不错的例子，原因在于以它为代表的数字信息的所有权，第一次引起了社会、经济和法律上的广泛关注。数字音乐与唱片或磁带不同，你可以在自己家的计算机里无限次地复制它，完全免费，而且还可以通过互联网把它复制发送到世界的任何角落，不会有任何音质损失，同样完全免费。唱片业把这当成了严重的威胁，试图通过法律或政治手段阻止数字音乐的拷贝。这场战争远未结束，每天都有小规模的战役打响，因此而对簿公堂和引爆政治辩论俨然成了家常便饭。

什么是声音？音源通过振动或快速运动引起空气压力的波动，人的耳朵把这种压力变化转换为神经活动，经大脑解释之后就形成了“声音”。1870 年代，托马斯·爱迪生制造了一个叫做“留声机”的机器，这台机器能把声波转换为蜡筒上类似的螺旋沟槽，而通过这些沟槽又能再次创造出同样的气压波动来。把声音转换为沟槽就是“录音”，而从沟槽换回到气压波动就是“回放”。爱迪生的发明迅速地得到改进，1940 年代就出现了密纹唱片（long-playing record）或简称 LP，而且至今还在使用（尽管数量已经不多了）。麦克风随着时间推移把变化的声压转换为变化的值并记录下来，然后根据

这些值在乙烯基的盘片上压制出与声压一致的螺旋沟槽。播放 LP 时，唱针随着沟槽起伏，其运动轨迹被转换为波动的电流，电流经过放大后驱动扬声器或耳机，通过它们的振动薄膜产生声音。

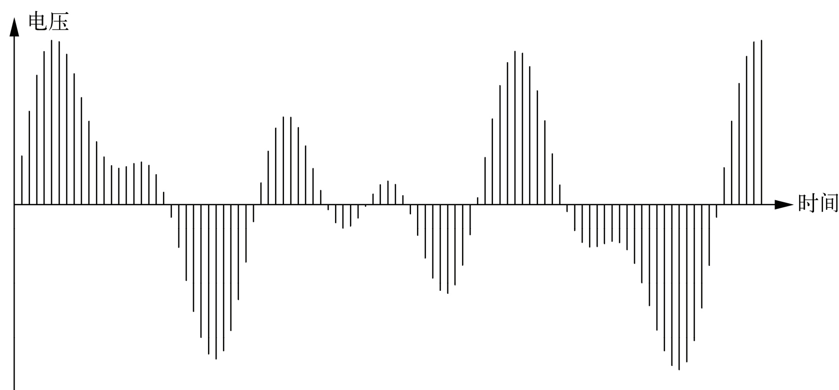


把空气压力随时间的变化形象地绘制出来并不难。其中压力可以用任何物理方法来表示，在此我们假设用电路中的电压。当然，电流、光的亮度，以及爱迪生发明的留声机中的纯机制装置都没有问题。



图中声波的高度表示声音强度或大小，水平方向的坐标轴表示时间：每秒钟声波的数量就是声调或频率。

假设我们以固定时间间隔连续测量这条曲线的高度（在这里就是电压值），就会得到下图所示的这些垂直线条。



测量得到的数值连接起来与曲线近似，测量越频繁，越准确，结果也就越吻合。测量得到的数值序列是波形的数字化表示，可以存储、复制、操作它们，也可以把它们发送到任何地方。如果有设备把这些数值转换成对应的电压或电流，然后再通过电压或电流驱动音箱或耳机，就能够实现回放。从声波到数值是模数转换，相应的设备叫 A/D 转换器；反过来当然是数模转换，或者叫 D/A。转换过程并不是完美无缺的，两个方向的转换都会损失一点信息。但大多数情况下，这种损失是人所觉察不到的（当然，也有不少唱片发烧友会说 CD 音质不如 LP 好）。

1982 年左右，音频光盘（或 CD）面世，成为最早承载数字声音的产品。今天的读者，恐怕没有谁不认识这个标志：



与 LP 唱片上的模拟沟槽不同，CD 用长长的螺旋状轨道在盘面的一侧记录数值。轨道上任意一个区块的表面要么平滑，要么是一个微小的凹坑。这些下凹或平滑的区块就是用来编码声波的数字值的，每个区块是一位，连续的多位表示二进制编码中的一个

数值，二进制的概念我们在下一节再介绍。光盘旋转时，一束激光照射到轨道上，而光电传感器则检测每个区块上反射回来的光量多少。如果光量不多，说明是凹坑；如果反射光很强，说明不是凹坑。标准 CD 编码采样率为每秒 44 100 次，而每次采样获得两个振幅值（立体声的左、右声道），精确度为 65 536（即 2^{16}，这并非巧合）分之一。轨道上的每个区块非常非常小，小到只有用显微镜才能看见，一张 CD 的表面上有 60 亿个小区块。（DVD 中的区块更小，由于区块更小，激光束频率更高，DVD 的存储容量近 5 GB，而 CD 大约为 700 MB。）

音频 CD 的出现几乎让 LP 没有了立足之地，相比之下，CD 的优点实在太多了：落上点灰尘也不用太担心了，更没有磨损一说，而且绝对小巧。但到了我写这本书的时候，LP 开始在某种程度上复苏，流行音乐 CD 的人气则日渐衰退。有朝一日，CD 很可能也会像 LP 一样变成古董，这倒让我很高兴，因为我收藏的音乐全部都是 CD 格式的。我现在完全拥有它们，而它们的存在将比我的生命更久远。CD 还有第二个用途，那就是作为存储、分发软件及数据的介质，不过这个功能已经被 DVD 取代，而 DVD 很可能又会被下载所取代。

声音和图片经常会被压缩，因为这两种媒体包含很多人类根本感知不到的细节。对于音乐，典型的压缩技术是 MP3，大约能把音频文件的体积压缩到原来的十分之一，同时几乎让人感觉不到音质下降。对于图片，最常用的压缩技术是 JPEG（是制定该标准的联合图像专家组——Joint Photographic Expert Group 的英文字头），它的压缩率也能达到 10 倍甚至更高。上文提到很多处理对数字信息能做，但对模拟信息却很难（或不可能），压缩就是一个例子。第 9 章我们再进一步探讨压缩。

那电影呢？1870 年代，摄影师埃德沃德·迈布里奇向世人证明，快速连续地显示一系列静态图片能够创造出运动的错觉。今天，电影显示影像的速度是每秒 24 帧，而电视大约是 25 到 30 帧，这个速度足以让人的眼睛把顺序播放的影像感知为动画。而通过组合（并同步）声音和影像，就可以创造出数字电影。而利用压缩技术减少空间占用，则催生了包括 MPEG（代表 Moving Picture Experts Group）在内的标准电影格式。实际上，视频的表示要比单纯的音频表示更复杂，一方面是它本身就复杂，另一方面很大程度上还因为它受到了电视的拖累，而电视在其存在的大部分时间内都是模拟的。模拟电视在世界范围内正逐渐被淘汰，而美国 2009 年已经将广播电视切换成了数字信号。

还有一些信息很方便以数字形式来表示，因为除了想好如何表示它之外，根本不需要做什么转换。比如这本书中的文字、字母、数字和标点符号，我们称为其普通文本。可以为其中每个字母指定一个唯一的数值，如 A 是 1，B 是 2 等等，这不就是一种数字化表示方法嘛。而事实也正是如此，只不过在表示标准中，A 到 Z 用的是 65 到 90，a 到 z 用的是 97 到 122，数字 0 到 9 用的是 48 到 57，而标点符号等其他字符用的是其他数值。这个表示标准叫做 ASCII，即 American Standard Code for Information Interchange（美国信息交换标准代码）。

下面给出了 ASCII 中的部分标准编码，前四行被我省略了，因为其中包含的都是些制表符、空格符等非打印字符。

32	space	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

不同地区有不同的字符集标准，但也有一个世界通用的标准叫 Unicode，它为所有语言的所有字符都规定了一个唯一的数值。这是一个非常庞大的字符集，人类的创造力是无穷无尽的，但在建立自身书写系统方面却很少有规则。目前，Unicode 涵盖的字符远远超过 100 000 个，而且这个数字还在稳步增长。可想而知，Unicode 中的大部分都是包括中文在内的亚洲字符集，但决不限于此。要了解 Unicode 都包含哪些字符集，可以访问 unicode.org，这个站点内容丰富，强烈推荐读者去看一看。

一言以蔽之：数字表示法能够表示上述所有信息，以及任何可以转换为数值的信息。因为只有数值，所以就可以用数字计算机来处理，而且正如第 9 章要介绍的，还可以通过互联网等通用数字网络将它复制到其他计算机上去。

2.3 比特、字节与二进制

“世界上只有 10 种人，理解二进制的和不理解二进制的。”

数字系统用数值来表示所有信息，但使用的却不是我们熟悉的（以 10 为基数的）十进制，这乍一看让人有点纳闷。那使用几进制？使用二进制，也就是逢 2 进 1 的数制。

虽说是人都或多或少懂点数学，但以我的经验来看，人们对数值的理解有时候并不靠谱。这一点从描述（再熟悉不过的）十进制和（多数人不熟悉的）二进制之间的关系就可以看出来。在这一节里，我会尽最大努力讲得通俗易懂，万一你听不懂或者我没讲清楚，你只要不断对自己重复一句话："跟普通的数一样，只不过是逢 2 而不是逢 10 进 1。"

2.3.1 比特

表示数字信息的最基本单位是比特（bit）。英文 bit 是合并 binary digit（二进制数字）之后造出来的，造这个词的人是统计学家约翰·图基，时间是 1940 年代中期。（图基还在 1958 年发明了单词 software——软件。）据说，鼎鼎大名的氢弹之父爱德华·泰勒更喜欢"bigit"这个词，但这个词最终没有流行起来，真是谢天谢地。binary 是指只有两个值的东西（前缀"bi"的意思就是"两个"），事实也的确如此：一个比特就是要么是 0 要么是 1 的一个数，没有其他可能。而十进制中有 0 到 9，共有 10 种可能的值。

只用一个比特，可以表示任何二选一的事物。这种二选一的例子比比皆是：开/关、真/假、是/否、高/低、进/出、上/下、左/右、南/北、东/西、男/女，等等。一个比特足以让人确定选择了两个中的哪一个。举个例子，我们可以用 0 表示"关"，用 1 表示"开"，或者反过来，哪个值表示哪个状态都无所谓，只要大家都没有意见就行。这张图是我的打印机的电源开关，由此可见，惯例还是用 0 表示"关"，用 1 表示"开"。



一个比特表示开/关、真/假之类的二选一的情形没有问题，但我们经常还要面对更多选项，表示更复杂的事物。为此，可以使用一组比特，然后为不同的 0 和 1 的组合赋予不同的含义。比如，可以用两个比特来表示大学四年：新生（00）、大二（01）、大三（10）和毕业班（11）。如果再多考虑一种情况，比如研究生，那两个比特就不够用了，因为两个比特只有 4 种组合，没有第五种可能。但是三个比特没问题，实际上三个比特能表示 8 种不同的情况，这样我们就可以把教师、教工和博士后都包含进来。三个比特的全部组合为：000、001、010、011、100、101、110 和 111。

比特数与它们所能表示的情况数之间有一个关系，很简单：N 个比特能表示 2^N 种组合，即 2×2×2×⋯×2（乘 N 次）。据此，就有：

比特数	值数	比特数	值数
1	2	6	64
2	4	7	128
3	8	8	256
4	16	9	512
5	32	10	1 024

对于十进制数呢？其实也有类似的关系：N 个十进制数字，可以表示 10^N 种不同的情况（我们称之为“数值”）：

位数	值数	位数	值数
1	10	6	1 000 000
2	100	7	10 000 000
3	1 000	8	100 000 000
4	10 000	9	1 000 000 000
5	100 000	10	10 000 000 000

2.3.2　2 的幂和 10 的幂

由于计算机中的一切都是以二进制形式来处理，因此像大小、容量等概念一般都是用 2 的几次幂来表达的。如果有 N 比特，那么就有 2^N 种可能的值，所以知道 2 的幂是多少（比如到 2^{10}）是很有用的。但随着数值越来越大，完全记住它们也没有什么必要。好在有一种简便的方法，可以得到它们的近似值：2 的某次幂与 10 的某次幂接近，它们的对应关系严格有序，容易记忆：

2^{10} = 1 024	10^3 = 1 000 (千，kilo)
2^{20} = 1 048 576	10^6 = 1 000 000 (兆，mega)
2^{30} = 1 073 741 824	10^9 = 1 000 000 000 (吉，giga)
2^{40} = 1 099 511 627 776	10^{12} = 1 000 000 000 000 (太，tera)
2^{50} = 1 125 899 906 842 624	10^{15} = 1 000 000 000 000 000 (拍，peta)
...	

（这个对照表最后包含的表示大小的单位叫“拍”或 10^{15}，其英文单词发音不是“皮”而是“拍”。另外，书后附有一个更全的词汇表，列出了更多单位。）随着数值增长，这个近似值的误差也会增大，不过到了 10^{15} 这么大的时候误差也就 12.6%，所以还是可以在很大范围内使用的。经常会有人混淆上述 2 的幂与 10 的幂之间的关系（有时候是想用来支持他们的观点），于是 kilo 或 1 K 可能是指 1000，但也可能指 2^{10} 即 1024。一般来说，这种混淆导致的误差并不大，因此在涉及很大的比特数时，用 2 和 10 的幂来做心算没什么问题。

说起来可能有点不值当的，但这种差异有时候也会带来麻烦。例如，著名硬盘驱动器制造商美国希捷公司，就曾因为这种差异在加州卷入一场集体诉讼。“原告称希捷公司对存储容量的术语 gigabyte（或 GB），使用的是十进制定义，即 1 GB = 10^9（1 000 000 000）字节，这是在误导消费者，因为计算机操作系统报告硬盘容量时，使用的是 GB 的二进制定义，即 1 GB=2^{30}（1 073 741 824）字节，相差大约 7%。”我个人并不认为很多消费者被严重误导了，但希捷公司选择了和解，并没有据理力争。

2.3.3　二进制数值

如果每个数字都能按照通常的进位法则来解释，那么一系列比特就可以表示一个数值，只不过此时的基数是 2，而不是 10。0 到 9，共 10 位数字，足以为 10 个项目计数或分配标签。如果数量超过 10，则必须使用更多位数字，比如两位十进制数字可以表示的数值或标签能达到 100 个，即 00 到 99。多于 100 的时候，就要用三位数字，其表示的范围是 1000，即 000 到 999。（根据约定俗成的做法，我们平时不会写出数值前导的零，但这些零是暗含的。另外，我们平时计数也都从 1 而非 0 开始。）

十进制数值实际上是 10 的某次幂之和的简写，比如 1867 就是 $1\times10^3+8\times10^2+6\times10^1+7\times10^0$，即 1×1000+8×100+6×10+7×1，即 1000+800+6+7。上小学的时候，你把

它们叫做个位、十位、百位……。这些概念我们太熟了，熟到根本不用去想。

二进制数也一样，只不过基数是 2，不是 10，而且只涉及 0 和 1 两个数字。比如可以把二进制数 11101 看成 $1\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$，用十进制来表示，就是 16+8+4+0+1，即 29。

既然比特序列可以解释为数值，那么自然就有了为项目分配二进制标签的模式：按数值顺序排列。前面我们看到了为新生、大二、大三、大四学生分配的标签 00、01、10、11，它们分别是十进制数值的 0、1、2、3。紧接着的序列是 000、001、010、011、100、101、110、111，也就是十进制数值 0 到 7。

下面我们做个练习，看你理解了多少。我们都熟悉扳着自己的手指从一数到十，要是你用二进制数来数（每个手指，包括大拇指，都代表一位二进制数），最大能数到多少？值的范围有多大？如果你数到 132，发现它的二进制表示是一个似曾相识的手势，那说明我前面讲的你都理解了。

前面我们都看到了，把二进制转换成十进制很容易：只要把相应位置上值为 1 的 2 的对应次幂加起来即可。而把十进制转换成二进制要难一些，但也不太难。就是反复地用 2 除十进制数。每次除完，把余数写下来，要么是 0，要么是 1，然后再用 2 除商。这样反复除下去，直到原来的数被除到等于 0。最后得到的余数的序列，就是相应的二进制数，但顺序相反，所以要倒转一次。

举个例子，把 1867 转换为二进制数的步骤如下：

十进制数	商	余数
1867	933	**1**
933	466	**1**
466	233	**0**
233	116	**1**
116	58	**0**
58	29	**0**
29	14	**1**
14	7	**0**
7	3	**1**
3	1	**1**
1	0	**1**

反向读取这些位，能得到 111 0100 1011，把相应位上为 1 的 2 的对应次幂加起来可以验算：1024+512+256+64+8+2+1 = 1867。

整个过程的每一步都会产生剩余数值的最低有效位（即最右边的位）。其实，把一个很大秒数表示的时间转换成日、时、分、秒的过程与此类似：除以 60 得到分钟（余数是秒），结果除以 60 得到小时（余数是分钟），结果再除以 24 得到天数（余数是小时）。区别在于时间转换使用了不止一个基数，而是先后用到了 60 和 24。

二进制的算术实在太简单了。因为总共才两个数字，加法和乘法表都只有两行两列。

+	0	1
0	0	1
1	1	0进1

×	0	1
0	0	0
1	0	1

虽然你将来不太可能自己动手做二进制算术，但这两个如此简单的表，其实也说明了为什么相对于十进制算术，执行这种计算的计算机电路要简单得多。

2.3.4 字节

在所有现代计算机中，数据处理及内存组织的基本单位都是 8 个比特。8 比特被称为 1 字节，而字节（byte）这个词是由 IBM 的计算机设计师维尔纳·巴克霍尔兹（Werner Buchholz）在 1956 年发明的。一个字节可以编码 256 个不同的值（2^8，即 8 个 0 和 1 的所有不同组合），这个值可以是一个 0 到 255 间的整数，也可以是 ASCII 字符集中的一个字符，或者其他什么。通常，为了表示更大或更复杂的数据，需要用到多个字节的字节组。两个字节有 16 比特，也就是 16 位，可以表示 0 到 $2^{16}-1$（65 535）之间的数值。两个字节也可以表示 Unicode 字符集中的一个字符，比如以下字符中的任意一个：

東京

这是两个字符，即“东京”，每个字符占两个字节。四个字节是 32 位，既可以表示“东京”，也可以表示最大直至 $2^{32}-1$ 的值，这个最大值大约是 43 亿。用一组字节表示什么都可以，但 CPU 自己特别定义了一些适中的字节组（比如表示不同大小的整数），以及处理这些字节组的指令。

要是你想把一或多个字节表示的数值写出来，那可以用十进制，如果它真是一个数值的话，十进制是最适合人类看的。如果你想看清每一个比特，特别是在不同比特编码不同信息的情况下，那还是二进制更方便。然而，二进制写起来太长了，比十进制格式长三倍还多，因此我们常用另一种替代数制，即十六进制。十六进制的基数是 16，因此也就有 16 个数字（就像十进制有 10 个数字，二进制有 2 个数字一样），分别是 0、1、…、9、A、B、C、D、E、F。每个十六进制数字表示 4 个比特，对于一般的数值，十六进制 0 相当于二进制 0000，依此类推，十六进制 9 相当于二进制 1001。接下去，十六进制 A 相当于二进制 1010（十进制 10），十六进制 B 相当于二进制 1011（十进制 11），依此类推，十六进制 F 相当于二进制 1111（十进制 15）。

除非你是程序员，否则能看到十六进制数的机会并不多。一个例子就是网页中的颜色值。前面说过，计算机中一个像素的颜色值大都使用三个字节来表示，一个表示红色分量，一个表示绿色分量，最后一个表示蓝色分量，这就是所谓的 RGB 编码。红绿蓝三个组分分别用一个字节表示，因此红色分量就有 256 种可能的值，三个组分中的绿色分量也有 256 种可能的值，同样，三个组分中的蓝色分量也有 256 种可能的值。于是一个像素可能的颜色值就是 256×256×256 种，听起来好多啊。我们可以用 2 和 10 的幂来简单估计一下这个数有多大。这个数是 $2^8\times2^8\times2^8$，即 2^{24} 或 $2^4\times2^{20}$，大约是 16×10^6，即 1600 万。在描述计算机显示器的情况下，你可能听说过这个数（“超过 1600 万种颜色！”）。

一个深红色的像素可以表示为 FF0000，换句话说，就是红色分量最多，没有绿色和蓝色；而一个鲜蓝色（并非深蓝色），即类似很多网页中链接的颜色，可以表示为 0000CC。黄色是红加绿，因此 FFFF00 就是最深的黄色。阴影的灰色具有等量的红、绿、蓝组分，因此一个中等灰度的像素应该是 808080，也就是红、绿、蓝组分的数量都相等。黑色和白色分别是 000000 和 FFFFFF。

Unicode 编码表就使用十六进制来表示字符：

東京

上面两个字符的十六进制编码为 6771 4EAC。第 8 章将会介绍以十六进制表示的以太网地址，第 10 章则会讨论用十六进制表示 URL 中的特殊字符。

有时候，在某计算机的广告中，我们会看到“64 位”这个说法（“Windows 7 家庭高级版 64 位”）。什么意思呢？计算机在内部操作数据时，是以不同大小的块为单位的，这些块包含数值（32 位和 64 位表示数值比较方便）和地址，而地址也就是信息在 RAM 中的位置。前面所说的 64 位，指就是地址。大约 25 年前，16 位地址升级到了 32 位地址（足够访问 4GB 的 RAM），而现在 32 位又升级到 64 位。我不想预测什么时候会从 64 变成 128，总得过上好一阵子吧，先不必想那么多。

关于比特和字节，我们讨论到现在最重要的是必须知道，一组比特的含义取决于它们的上下文，光看这些比特看不出来。一个字节可以只用 1 个比特来表示男或女，另外 7 个空闲不用，也可以用来保存一个不大的整数，或者一个#之类的 ASCII 字符，它还可能是另一种书写系统中一个字符的一部分，或者用 2、4 或 8 个字节表示的一个大数的一部分，一张照片或一段音乐的一部分，甚至是供 CPU 执行的一条指令的一部分。

事实上，一个程序的指令就是另一个程序的数据。从网上下载一个新程序，或者从 CD-ROM 或 DVD 中安装该程序时，它就是数据，所有比特将无一例外地被复制一遍。但在运行这个程序时，它的比特会被当成指令，CPU 在处理这些比特时，又会把它们当成数据。

2.4 小结

为什么用二进制而不用十进制？因为制造只有两种状态（如开和关）的物理设备，比制造有十种状态的设备更容易。这种简单的性质在数不清的技术中都得到了利用，比如：电流（流动或不流动）、电压（高或低）、电荷（存在或不存在）、磁性（南或北）、光（亮或暗）、反射率（反光或不反光）。约翰·冯·诺依曼很早就清楚地认识到了这一点，他在 1946 年说过：“我们储存器中最基本的单位自然是采用二进制系统，因为我们不打算度量电荷的不同级别。”

为什么我们要知道或者要关心二进制数呢？这个问题问得好。至少在我的课上，理解另一种不熟悉的数制，相当于做了一次量化推理的练习，而有了这个训练之后，对我们习以为常的十进制的理解也将更深一层。除此之外，另一个意义在于，比特的数量

在一定程度上揭示了涉及的空间、时间或者复杂性。再从根本上说，计算机值得我们花时间去理解，而二进制正是其运作的核心所在。

现实生活中也能找到一些与计算机无关的应用二进制的场景，或许是因为人们都认为大小、长短的加倍、减半是一种自然而然的运算。比如，高德纳在《计算机程序设计艺术》中描述了 14 世纪英国的酒器单位，分为 13 个二进制量级：2 吉耳是 1 超品（chopin），2 超品是 1 品脱，2 品脱是 1 夸脱，依此类推，直到 2 百瑞尔（barrel）是 1 豪格海（hogshead），2 豪格海是 1 派普（pipe），2 派普是 1 坦恩（tun）。这些单位中差不多还有一半仍然在英制液体度量体系中使用。当然，其中一些很令人陶醉的词，比如费尔金（firkin）和基尔德坎（kilderki）（2 费尔金是 1 百瑞尔），今天已经很难得见了。

第 3 章

深入了解 CPU

第 1 章我们说过，中央处理单元或者 CPU 是计算机的“大脑”，但当时并没有就这个术语多作讨论。这一章我们就来详细探讨一下 CPU，因为它是一台计算机中最重要的组件，而且理解其特性对理解本书后面的内容也至关重要。

中央处理器如何工作？ 它处理什么，怎么处理？直观来讲，CPU 有一个小型指令系统，包含着它能够执行的基本操作。它可以做算术题，加、减、乘、除，跟计算器一样。它可以从 RAM 中取得要操作的数据，然后再把结果保存到 RAM，与很多计算器中的存储操作一样。CPU 还要控制计算机的其他组件，确保鼠标、键盘等外围设备输入的数据得到响应，让信息在屏幕上得以显示，同时还要控制和协调连接到计算机的其他所有器件。

最重要的是，它可以作出决定——尽管是简单的决定：它可以比较数值（这个数比那个数大吗？）或者比较其他数据（这段信息与那段信息一样吗？），还能根据结果决定接下来做什么。这一条最重要，因为这意味着 CPU 能做的虽然比计算器多不了多少，但它可以在无人看管的情况下完成自己的工作。正如冯・诺依曼所说的：“要让这种机器完全自动化，即让它在计算开始后不再依赖人工操作。”

由于 CPU 能根据它所处理的数据决定下一步做什么，因此它就能自己运行整个系统。虽然其指令系统并不大，或者说并不复杂，但 CPU 每秒可以执行数十亿次运算，所以它能完成极为复杂的处理。

3.1　玩具计算机

为了解释 CPU 如何工作，我们先介绍一台不存在的机器。这是一台编造的或者说“假想的”计算机，它与真正计算机的原理相同，只不过要简单得多。因为这台计算机只存在于纸面上，所以我就可以随意设计它，让它能有助于我们理解计算机的工作机制。我还可以写一个真正的计算机程序，用它模拟我的纸面设计，这样就可以给假想的机器写程序，看看这些程序怎么运行。

我想了一下，就管这个编造的机器叫“玩具”计算机吧，因为它不是真的，但又具有真正计算机的很多特性。实际上，它跟 1960 年代末的小型机差不多是一个水平，某种程度上与冯·诺依曼论文中的例子相近。这个玩具有用来存储指令和数据的 RAM，还有一块额外的存储区叫**累加器**，其容量足以存储一个数值。累加器类似于计算器的显示屏，保存用户最近输入的数值，或者最近计算的结果。玩具还有一个指令表，只包含 10 个指令，都是前面提到过的基本操作。比如下面几个指令：

```
GET         从键盘获取数值并放到累加器中
PRINT       打印累加器中的内容
STORE M     把累加器中内容的副本保存到位置 M（累加器的内容不变）
LOAD Val    把 Val 加载到累加器，Val 是一个值或值在存储器中的位置（Val 不变）
ADD Val     把 Val 与累加器中的内容相加（Val 不变）
STOP        停止运行
```

为了保证最简单，我们规定每个 RAM 位置可以保存一个数值或一条指令（至于如何表示先不说），这样就可以在 RAM 中保存由指令和数据共同组成的程序。运行时，CPU 从 RAM 的第一个位置开始，重复如下简单的循环：

```
读取  从 RAM 中取得下一条指令
译码  搞明白该指令要做什么
执行  执行指令
      返回“读取”
```

而给这个“玩具”计算机编写程序，就是要写出一组完成相应任务的指令，把它们放到 RAM 中，然后告诉 CPU 去执行这些指令。举个例子，比如 RAM 中正有如下指令（指令在 RAM 中都保存为等价的数值形式）：

```
GET
```

```
PRINT
STOP
```

运行程序时，第一条指令会要求用户输入一个数值，第二条指令会把该数值打印出来，而第三条指令告诉处理器停止执行。这个过程似乎很无聊，但却足以说明程序是如何运行的。假如真有这么个“玩具”计算机，这个程序还真能运行哩。巧啦，确实有这种“玩具”计算机，下面就是其中一个在运行时的样子了：

Program: Output:

GET
PRINT
STOP

RUN Accumulator:

按下 RUN，出现一个对话框：

[JavaScript Application]

Enter value for GET

Cancel OK

键入一个数字，按下 OK，结果就这样了：

Program: Output:

GET
PRINT
STOP

123

stopped

RUN Accumulator: 123

跟我们期望的一样，程序要求输入数值，打印它，然后停下来。

再看一个复杂些的程序。这个程序多了向 RAM 中存储值，然后再取出的操作。具体来说，程序先把一个数读到累加器中，然后把这个数保存到存储器，再把另一个数读到累加器中（覆盖前一个数），给它加上第一个数（从 RAM 中保存该数的位置取出），打印两个数的和，然后停下来：

```
     GET         取得第一个数并放到累加器中
     STORE Mem   把这个数保存到 RAM 中的位置 Mem
     GET         取得第二个数并放到累加器中
     ADD Mem     给它加上第一个数
     PRINT       打印两个数的和
     STOP
   Mem  -        RAM 中的一个位置，保存用户输入的第一个数值
```

CPU 从程序的起点开始，每次取得一条指令。执行完一条指令后，继续取得并执行下一条指令。

唯一麻烦点儿的地方是要在 RAM 中辟出一个位置，好保存读到的第一个值。但不能把这个值留在累加器中，因为第二个 GET 指令会覆盖它。由于它是数据，而非指令，因此必须把它放在 RAM 中一个不会被当成指令取出来的位置。把数据放到程序末尾，所有指令后面，CPU 就不会把它解释成指令了，因为 STOP 指令会让 CPU 停下来，不会继续读取下一个位置。

还需要一种引用该位置的方法，这样程序中的指令才能在必要时找到它。或许可以把它放在存储器的第七个位置（即第六条指令后面），但这样就要先数一数有多少条指令，很麻烦，而且一旦修改了程序，位置可能也要随之变更。通行的做法是给该数据项起个名字（第 5 章也会讨论），让一个程序负责跟踪记录该数据项在 RAM 中的实际位置，然后用实际的位置代替名字。名字可以任意起，但最好是起一个让人一看就明白相应数据或指令含义的名字。比如，这里的 Mem，其实改为 Firstnum 更好理解。

怎么扩展这个程序，让它能计算三个数的和呢？没错，可以再加一组 STORE、GET 和 ADD 指令（有两个位置可以插入这组指令），但这种方法肯定不能扩展到 1000 个数相加。并且，在事先不知道有多少数的情况下也行不通。

答案是给 CPU 的指令表增加一个新指令，通过它能重用已有的指令序列。这就是 GOTO 指令，有时候也叫“分支”或“跳转”，它告诉 CPU 读取下一条指令时不要从序列中的下一个位置读，而要从它指定的位置读。

使用 GOTO 指令，可以让 CPU 返回到程序的前面，重复执行指令。举一个简单的例子，比如写一个程序随时显示每个输入的值。这个功能正是数据复制程序的核心，也能说明 GOTO 指令的作用。我给这个程序的第一条指令加了个标签，叫 Top（能说明其角色的任何名字都行），而最后一条指令告诉 CPU 再返回到程序的第一条指令：

```
Top  GET           取得一个数并放到累加器中
     PRINT         打印出来
     GOTO Top      返回 Top，取得另一个数
```

问题只解决了一半：可以重用指令，但却没办法停止循环。为此，还得再增加一条指令，该指令先测试一个条件，然后决定接下来该做什么，而不是一味往前走。这样的指令被称为"条件分支"指令，或条件跳转、条件转向。现实中，不同的机器在实现该指令时测试的条件会有所不同，但有一种可能是测试某个值是否等于零，如果是则跳到某个位置。那好，我们就给这个"玩具"计算机的指令表添加一个 IFZERO 指令，它会在累加器的值等于零的时候把 CPU 引导到一个指定的位置，否则继续顺序执行下一条指令。下面这个程序就使用了 IFZERO，该程序会在输入值不等于零的情况下不断打印输入的值：

```
Top  GET           取得一个数并放到累加器中
     IFZERO Bot    如果累加器的值为零，跳到标签为 Bot 的指令
     PRINT         值不是零，打印出来
     GOTO Top      返回 Top，取得另一个数
Bot  STOP
```

只要用户不觉得无聊，这个程序就会不断取得数据并打印出来。如果用户输入了零，程序就会跳到 STOP 指令（标签 Bot 代表"bottom"，即"末尾"的意思）并退出。

注意，程序不会打印那个表示终止输入的零。怎么修改一下程序，让它能打印出这个零再停止呢？这个问题不难——答案明摆着——但它却足以揭示一个现象：简单地交换两个指令的位置，就能导致程序的行为与我们的设想出现偏差，或者说会导致程序去做一些与我们希望的完全不同的事。

组合使用 GOTO 和 IFZERO，可以写出在指定条件为真之前可以重复执行指令的程序，而 CPU 也可以根据之前计算的结果改变计算过程。指令表中有了 IFZERO 之后，我们的玩具 CPU 可以执行**任何**计算。尽管有点难以想象，但这些指令足以应对数字计算机能够完成的任何计算——任何计算都可以分解为能使用基本指令完成的小步骤。

关于任务分解的思想，我们会随时随地提到，因为它太重要了。（想一想，既然有了 IFZERO，那严格来讲 GOTO 还有必要吗？能不能用前者来模拟后者呢？）

例如，下面这个程序可以把一大堆数加起来，到输入了零为止。使用特殊的值终止循环输入是一种常见的做法。因为对于求几个数之和的计算，加零没有意义，所以这里使用零很合适。

```
Top  GET            取得一个数
     IFZERO Bot     如果这个数是零，转到 Bot
     ADD Sum        把累计的和与这个数相加
     STORE Sum      把结果存储为一个新的累计和
     GOTO Top       返回 Top 再取得另一个数
Bot  LOAD Sum       把累计和加载到累加器
     PRINT          然后打印出来
     STOP
Sum  0              保存累计和的存储器位置（程序启动时的初始值为 0）
```

玩具模拟器对这个程序最后一行的“指令”是这样解读的：给一个存储器位置命名，然后在程序运行前放进去一个值。之所以需要这样做，是因为程序要计算很多数的和，而且累计和的初始值必须从零开始。

怎么检验这个程序，确定它没问题呢？表面上看没有问题，但有些问题很容易被忽视，所以有必要进行系统的测试。注意这里说的是“系统的测试”，并非随便给程序输入几个值。

最简单的测试用例是什么？如果根本没有要相加的数，除了用来终止输入的零，则和应该为零，这就是第一个测试用例。然后要试试只输入一个数，而和就应该是那个数。接下来可以试试两个数，这两个数的和你是知道的，比如 1 加 2 等于 3。此时，基本上就可以确定程序没有问题了。假如你真用心的话，通过一步一步地仔细过几遍指令，就可以在程序上机之前完成对代码的测试。优秀的程序员无论写什么都会自己先做这种检查。

到目前为止，我们一直没有讨论指令和数据在 RAM 中是如何表示的。说一种可能性吧，比如每条指令都需要一个存储器的位置存储其数值代码，而在该指令引用存储器位置或有一个数据值的情况下，还需要另一个紧随其后的位置。也就是说，IFZERO 和 ADD 等指令因为引用了存储器位置，所以要占用两个存储器单元，其中第二个单元中保存的是它引用的位置；同样假设数据值也要占用一个位置。这是一种简化，但

实际情况也差不太多。最后，假设各个指令的数值代码（按它们在前几页中出现的先后顺序）分别为：GET=1、PRINT=2、STORE=3、LOAD=4、ADD=5、STOP=6、GOTO=7、IFZERO=8。

现在，我们来看看前面计算一系列数之和的程序。在程序刚开始运行的时候，RAM 中的内容如下图所示。图中也给出了为三个存储器位置指定的名字，给出了指令以及存储器单元的地址。

Top: GET	IFZERO	Bot	ADD	Sum	STORE	Sum	GOTO	Top	Bot: LOAD	Sum	PRINT	STOP	Sum:
1	8	10	5	14	3	14	7	1	4	14	2	6	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14

前面的玩具模拟器是使用 JavaScript 写的，第 7 章将介绍 JavaScript。扩展这个模拟器很容易，就算你以前从来没见过计算机程序，也可以给它添加一条乘法指令，或者其他的条件分支指令。

3.2 真正的 CPU

CPU 反复执行简单的循环：从存储器中取得下一条指令，该指令正常情况下保存在存储器的下一个位置，但也可以是使用 GOTO 或 IFZERO 指定的位置；对指令进行译码，也就是搞清楚这条指令要干什么，然后为执行该指令做好准备；执行指令，从存储器中取得信息，完成算术或逻辑运算，保存结果，总之是执行与指令匹配的组合操作；然后再从头取得指令，开始下一次循环。真正的处理器也执行同样的“取指令—译码—执行”循环，只不过为了加快处理速度，还会配备精心设计的各种机制。但核心只有循环，与前面重复把数值加起来的例子一样。

真正计算机的指令比我们玩具计算机的多，但性质相同。比如，有更多移动数据的指令，更多完成算术运算及操作不同大小和类型数值的指令，更多比较和分支指令，以及控制计算机其他组件的指令。典型的 CPU 有几十到数百个不同的指令；指令和数据通常要占用多个内存位置，通常为 2 至 8 个字节。真正的处理器有多个累加器，通常是 16 或 32 个，所以可以保存多个中间结果，而且都是速度极快的存储器。真正的程序与我们的玩具示例相比可谓庞大，有的甚至多达数百万条指令。至于如何编写这

么大的程序，本书后面有关软件的章节会详细讨论。

计算机体系结构是研究 CPU 与其他计算机组件连接的一门学科。在大学里，它通常是计算机科学和电子工程的交叉领域。

计算机体系结构研究的一个问题是指令集，也就是处理器配备的指令表。是设计较多的指令去处理各式各样的计算，还是设计较少的指令以简化制造并提升速度？体系结构涉及复杂的权衡，要综合考虑功能、速度、复杂性、可编程能力（如果太复杂，程序员将无法利用其功能）、电源消耗及其他问题。用冯 · 诺依曼的话说："一般来讲，运算器内在的经济性取决于期望的机器运行速度……与期望的简易性或低价位之间的折中。"

CPU 与 RAM 和计算机的其他组件是如何连接的？处理器非常快，通常执行一条指令只需要零点几纳秒。（1 纳秒等于十亿分之一秒，或者 10^{-9} 秒。）相对而言，RAM 则慢得让人难以忍受——从存储器中取得数据或指令大概要花 25 到 50 纳秒。当然，这里的快指的是绝对速度，而慢则是相对于 CPU 而言。假如 CPU 不必等待数据，那它可能早就执行完上百条指令了。

现代计算机会在 CPU 和 RAM 之间使用少量的高速存储器来保存最近使用过的指令和数据，这种高速存储器叫作**缓存**。如果可以从缓存中找到信息，那么就会比等待 RAM 返回数据快得多。下一节我们会详细介绍缓存及缓存机制。

设计师在设计体系结构的时候也有一套方法，能够让处理器跑得更快。比如，可以把 CPU 设计为交替地取得和执行指令，而同一时刻会有几个指令处于执行过程的不同阶段，这种设计叫做流水线。（与汽车装配线很相似。）结果呢，虽然某个特定的指令仍旧要花同样的时间完成，但其他指令都有机会得到处理，从整体上看完成这些指令则会快很多。另一种方法是并行执行多条互不干扰、互不依赖的指令，就相当于多条平行的汽车装配线。有时候，只要指令的操作不会相互影响，甚至可以不按顺序执行。

另外一种可能是同时运行多个 CPU。今天的笔记本电脑，甚至连手机都已经有多个 CPU 了。英特尔酷睿双核处理器在一块集成电路芯片上集成了两个 CPU（"核心"）。在一块芯片上集成越来越多的处理器已经成为明显的趋势。由于集成电路特征尺寸越来越小，因而可以集成在一块芯片上的晶体管数量必将越来越多，这些晶体管可以构成更多 CPU，也可以构成更多缓存。

处理器应用的领域决定了设计者要权衡哪些要素。很长时间以来，处理器主要的应用领域是桌面计算机，而桌面环境下的电源和物理空间都比较充足。因此，设计者只要专注于让处理器尽可能地快就好了，电源是用之不竭的，而散热只要多加风扇就行。笔记本电脑要求的权衡要素有了明显不同，一方面空间有限，另一方面在不插电的情况下，笔记本要靠沉重又昂贵的电池供电。其他方面条件不变，笔记本处理器必然要相对慢一些，耗电少一些。

手机和其他超轻便设备进一步提高了设计要求，因为尺寸、重量和电源各方面都有了更多限制。此时，单靠小范围调整设计是行不通的。虽然英特尔是台式机和笔记本处理器的主要供应商，但几乎所有的手机都使用"ARM"处理器，因为它耗电更少。ARM 处理器是指获得英国 ARM Holdings 公司许可制造的处理器。

比较不同 CPU 的速度并不是特别有意义。即便是最基本的算术运算，其处理方式也可以完全不同，很难直接比较。比如，同样是计算两个数的和并保存结果，有的处理器需要用三个指令（比如我们的玩具计算机），有的则需要两个，而有的可能只需要一个。有的 CPU 具有并行处理能力，或者说能够同时执行多条指令，从而让这些指令在不同阶段上执行。为了降低处理器的耗电量，牺牲执行速度，甚至根据是不是电池供电动态调整速度都是很常见的。对于某个处理器比另一个处理器"更快"的说法，不必太当真，因为很多情况下都要具体问题具体分析。

3.3 缓存

说到这里，有必要花点时间简单介绍一下缓存，这是一个在计算领域中广泛适用的思想。在 CPU 中，**缓存**是容量小但速度快的存储器，用于存储最近使用的信息，以避免访问 RAM。通常，CPU 会在短时间内连续多次访问某些数据和指令。例如，加法计算程序循环体中的多条指令，对每个输入值都要执行一遍。如果这些指令存储在缓存中，就不用在每次循环时都从 RAM 读取它们，这会让程序的速度快 50 倍。类似地，把 `Sum` 存储在数据缓存中也能提高访问速度。

典型的 CPU 有两到三个缓存，容量依次增大，但速度递减，一般称为一级缓存、二

级缓存和三级缓存。最大的缓存能存储以兆字节计的数据（我的 Macbook 有 3 MB 的二级缓存），大多数 CPU 的指令和数据缓存都是独立的。缓存之所以有用，关键在于最近用过的信息很可能再次被用到，而把它们存储在缓存里就意味着减少对 RAM 的等待。缓存通常会一次性加载一组信息块，比如只请求一个字节，但会加载 RAM 中一段连续的地址。因为相邻的信息也可能被用到，要用的时候它们同样已经在缓存里了，换句话说，对邻近信息的引用也不需要等待。

除了发现性能提升之外，用户是感受不到这种缓存的。但缓存的思想却无处不在，只要你现在用到的东西不久还会用到，或者可能会用到与之邻近的东西，那运用缓存思维就没错。CPU 中的多个累加器本质上也是一种缓存，只不过是高速缓存而已。RAM 也可以作为磁盘的缓存，而 RAM 和磁盘又都可以作为网络数据的缓存。计算机网络经常会利用缓存加速访问远程服务器，而服务器本身也有缓存。

在使用浏览器上网的时候，你可能见过“清空缓存”的字眼。对网页中的图片和其他体积较大的资源，浏览器会在本地保存一份副本，因为再次访问同一网页时，使用本地副本比重新下载速度快。缓存不能无限地增长，因此浏览器会悄悄地删除旧项目，以腾出空间给新的，它还给你提供了删除所有缓存内容的命令。

你自己随时可以检验缓存的效果。比如可以做下面两个实验，一是打开 Word 或 Firefox 等大程序，看看从启动到加载完成并可以使用要花多长时间。然后退出程序，立即重新启动它。正常情况下，第二次启动的速度会明显加快，因为程序的指令还在 RAM 里，而 RAM 正在充当磁盘的缓存。使用其他程序一段时间后，RAM 里会填满该程序的指令和数据，原先的程序就会从缓存中被删除。

二是在谷歌里搜索几个不太常见的单词或短语，注意谷歌查询结果要花多长时间。接着再搜索同样的关键词。返回搜索结果的时间会明显缩短，因为谷歌已经在其服务器上缓存了搜索结果。这个缓存对其他搜索相同关键词的人也有好处，因为缓存在谷歌的服务器上，不在你的机器里。要验证缓存在谷歌服务器上，可以在你搜索完之后，让别人在他们自己的计算机上搜索同样的关键词。虽然不能完全保证，但一般来说第二次搜索速度会快很多。

3.4 其他计算机

人们很容易认为计算机不是 PC 就是 Mac，因为那是我们最常见到的。实际上，还有很多其他类型的计算机。这些计算机无论大小，都具有相同的核心特性，即都能完成逻辑运算，并且都具有类似的体系结构，只不过在设计的时候会不同程度地考虑成本、供电、大小、速度等因素。手机和平板电脑也是计算机，它们运行操作系统并支持更加丰富多样的运算环境。比这还小的系统是嵌入式系统，日常生活里能见到的几乎所有数字设备里都有嵌入式系统，比如数码相机、摄像机、GPS 导航系统、家电、游戏机，等等 。

更大的计算机在很多年前就已经实现多个 CPU 共享内存了。如果能把大任务分解成小任务，而分解后的小任务又可以通过不同 CPU 协作完成，CPU 相互之间不会出现太长的等待，也不会有太多的相互干扰，那么就能以这种方式加快完成大任务。除了在大型系统中广泛应用，这种集成多个处理器的多核芯片在个人计算机中也已经司空见惯，而且未来很可能会普及。

超级计算机往往有大量的处理器和大量的内存，这些处理器本身可能带有一些特殊指令，在处理某种数据时，它们比通用的处理器速度更快。今天的超级计算机通常是高速计算机集群，CPU 仍然是普通的 CPU，并没有什么特殊的硬件。网站 top500.org 每六个月就重新公布一次全世界最快的 500 台计算机。最快速度的纪录不断被打破，几年前还能跻身排行榜前几名的计算机，今天可能已经在榜单上找不到了。2011 年 6 月最快的计算机有 50 多万个 CPU，每秒可以执行 8×10^{15} 次数学运算。

分布式计算指的是很多更加独立的计算机（比如不共享内存），而且地理上更加分散，甚至位于世界的不同地方。这样一来，通信更加成为瓶颈，但却能够实现计算机之间的远距离协作。大规模的 Web 服务，比如搜索引擎、在线商店和社交网络，都是分布式计算系统。在这种系统中，数以千计的计算机协作，可以为海量用户迅速地提供结果。

所有这些计算系统都有相同的基本原理。它们都使用通用处理器，可以通过编程完成无穷无尽种任务。每个处理器都有一个有限的简单指令表，能够完成算术运算、比较数据、基于前置计算结果选择下一条指令。不管物理结构的变化让人多么眼花缭乱，它们的一般体系结构从 1940 年代至今并没有太大的变化。

或许很难想象，这些计算机都具有相同的逻辑功能，可以完成一模一样的计算（暂且不论对速度和内存的要求）。1930 年代，这个结果就已经被几个人分别独立地证明过，其中包括英国数学家艾伦·图灵。对于非专业人员，图灵的手段最容易理解。他描述了一个非常简单的计算机（比我们的玩具计算机还简单），展示了它能够计算任何可以计算的任务。他描述的这种计算机，我们今天叫做**图灵机**。然后，他展示了如何创建一种图灵机，模拟其他图灵机，这种图灵机现在被称为**通用图灵机**。写一个模拟通用图灵机的程序很容易，而写一个程序让通用图灵机模拟真实的计算机也是可能的（尽管不容易）。实际上，从能够计算什么的角度讲，所有计算机都是等价的，尽管运行速度明显不可能等价。

第二次世界大战期间，图灵从理论转到实践：他领导开发了用于破译德军情报的计算机。1950 年，他发表了一篇名为“计算机器与智能”（Computing machinery and intelligence）的论文，其中提出一个测试（即今天所谓的**图灵测试**），人们可以通过该测试来评估计算机是否能表现出人类的智能。想象一下，一台计算机和一个人，通过键盘和显示器与另一个提问者交流。通过问答，提问者能确定哪个是人，哪个是计算机吗？图灵的想法是，如果不能明显地将二者区分开，那么计算机就表现出了智能的行为。

缩写词 CAPTCHA 中包含图灵的名字，这个缩写词代表“Completely Automated Public Turing test to tell Computers and Humans Apart”（用以区分计算机和人的完全自动化的公共图灵测试）。CAPTCHA（可以理解为“验证码”）就是一些扭曲变形的字母，广泛用于验证网站的用户是人而非程序：

following　　finding

CAPTCHA 是一个**反向图灵测试**，因为它利用了人比计算机更擅长识别文字这一特点，来达到区分人和计算机的目的。

图灵是计算机领域最重要的人物之一，他对人类理解计算做出了重大贡献。计算机科学领域的诺贝尔奖——图灵奖，就是以图灵的名字命名的。后面几章将陆续介绍一些获得过图灵奖的重要计算机发明。

硬件部分小结

关于硬件的讨论结束了，但后面偶尔还是会提到一些装置或设备。以下是你应该通过这一部分掌握的基本概念。

数字计算机包含处理器和存储器。处理器执行简单的指令，速度非常快。它可以根据早先计算的结果以及外界的输入，决定接下来做什么。存储器包含数据和处理数据的指令。

计算机是一种通用的机器。它从存储器中读取指令，而人把不同的指令放到存储器中，可以改变它要执行的计算。指令和数据要通过使用场景区分，一个人的指令可以是另一个人的数据。

图灵的结论：从能够执行完全相同的计算的意义上说，这种结构的所有计算机（包括你以后可能会看到的任何计算机）具有完全相同的计算能力。当然，它们的性能可能千差万别，但在不考虑速度和存储器容量的前提下，它们的能力则是等价的。最小最简单的计算机也能够完成大计算机所能完成的计算。的确，可以通过编程让任何计算机模拟其他计算机，而图灵正是这样证明了他的结论。

计算机的逻辑结构自冯·诺依曼之后并没有太大改变，但物理结构已经发生了巨大变化。摩尔定律已经应验了大约 50 年，成为迄今为止几乎完全兑现的预言。摩尔定律预言了在既定的空间和成本之下，个别器件的大小和价格会呈指数级下降，而它们的计算能力呈指数级增长。

还应该提到一个最重要的主题，即它们都是**数字**计算机：所有一切最终都要化简为比特，单独或成组地以数字形式表示信息，所谓信息可能是指令也可能是数据。这些比特的含义取决于它们的上下文。可以化简为比特的任何事物，都可以通过数字计算机来重现和处理。

海量的数据让计算机有可能代替人去完成一些通常只有人才能胜任的工作。不同语言间的翻译就是一个非常明显的例子。有这么多平行文本，从一种语言翻译到另一种语言，是一个聪明的程序完全可以做到的，尽管它们都还很不完善，但表现得却越来越好。以下是通过 Google Translate 把上面一段的英文翻译成法语的结果：

> （英语）These are digital computers: everything is ultimately reduced to bits, which individually or in groups represent information, whether instructions or data, as numbers. The interpretation of the bits depends on the context. Anything that we are able to reduce to bits can be represented and processed by a digital computer.

> （法语）Ce sont des ordinateurs numériques: tout est finalement réduit en morceaux, qui, individuellement ou en groupes représentent des informations, si les instructions ou des données, comme des nombres. L'interprétation des bits dépend du contexte. Tout ce que nous sommes en mesure de réduire de bits peuvent être représentées et traitées par un ordinateur numérique.

再翻译回英语：

> These are digital computers, everything is finally reduced to pieces which, individually or in groups that represent such information, either instructions or data. The interpretation of bits depends on the context. All that we are able to cut in pieces may be represented and processed by a digital computer.

从某种角度讲，互译结果相当不错，甚至还考虑到了一些语法问题。然而，翻译过程忽略了一个谈论计算机时至关重要的事实，即“比特”（bit）并不等同于“碎块”（piece），而“化简为比特”（reduced to bits）与“切分成碎块”（cut in pieces）的意思也完全不同。类似的结论同样适用于今天的语音识别、面部识别及其他图像处理：计算机可以做得很出色，但没人会说它们的能力可以与人类比肩。顺便说一下，如果你再试一次，

翻译的结果又会不一样，其算法和底层的数据好像变化很频繁。

计算机可以下出特级大师水平的国际象棋，但在人脸识别上却不及幼儿。早些时候，计算机下国际象棋的水平其实很烂，但随着机器的速度不断加快，它下棋的水平明显提高，但这一切几乎完全是因为它比人类对手能够多看几步。类似地，语言翻译、语音识别等领域最大的进步，主要还体现在庞大的数据量（比如不同语言的平行文本数量），而有了这些数据计算机才能接近人的表现。

2011 年 2 月，广受欢迎的电视节目 Jeopardy 发起了最强的两位人类参赛者与 IBM 的 Watson 计算机系统之间的对决。Jeopardy 智力问答需要掌握大量的事实性知识，还要求能够处理双关语、省略句、松关联，以及具备其他语言能力。公平地说，Watson 彻底打败了人类。图灵测试想做的，就是在隐身、忽略响应时间等因素的前提下，通过对人和计算机提问，看看能否区分哪个是人，哪个是计算机。阅读 Jeopardy 电视节目的文字版，你很难区分人类和计算机。

尽管如此，我们还是有太多太多的事物不知道怎么用比特来表示，更不必说怎么用计算机来处理了。比如，日常生活中最重要的一些事物：艺术、创意力、真理、美、爱、荣誉和价值。我想在一定的时期内，这些事物仍将超出计算机的能力之外。如果你碰见了一个人，声称知道怎样“通过计算机”处理这些东西，那你可不要随随便便就相信他。

第二部分

软　件

好消息是，计算机是一种通用机器，能够执行任何计算。虽然它只有很少的指令，但执行这些指令的速度却极快，而且它能够很大程度上控制自己的运行。

坏消息是，如果没有人告诉它该做什么，它就什么都不会做，而且得事无巨细一五一十地告诉它。计算机是“魔法师的学徒”，能够不知疲倦、分毫不差地执行指令，但下达给它的任务书也必须高度精确。

能够让计算机完成某种任务的指令序列通称软件。软件的“软”与硬件的“硬”相对，寓意看不见，摸不着。硬件是有形的：如果失手把计算机掉在脚上，你会喊疼。软件则没有这个问题。

在接下来的几章中，我们要讨论软件，即如何告诉计算机做什么。第 4 章会概括地谈谈软件，并着重讨论一下算法，它们实际上是诸多焦点任务的理想化解决方案。第 5 章讨论编程和编程语言，我们用它来表达一系列计算步骤。第 6 章介绍主要的软件系统，无论你知道与否，反正每天都在用。本部分最后一章是第 7 章，讲讲 JavaScript 编程。

在此期间，要把这些牢记于心：现代系统越来越多地采用通用硬件（如处理器、内存，以及与外界相连接的接口），同时靠软件来实现特定的行为。人们普遍认为，软件更便宜、更灵活，比硬件更好修改（特别是跟已经出厂的设备比）。例如，如果用一台计算机来控制汽车的动力和刹车，那么防抱死和电子稳定控制显然应该是软件的功

能。举一个明显的例子。2010 年 4 月的美国《消费者报告》(*Consumer Reports*)称丰田雷克萨斯 GX460 车型“不能买:存在安全隐患”,因为其电子稳定控制系统会导致这款 SUV 在急速转弯时车尾过分向外甩,从而可能导致翻车事故。一个月之内,丰田公司就升级软件,修复了这个问题。根本就没有任何机械问题,仅仅是软件最初有些问题。

此外,丰田车主必须到经销商处升级软件,到了那儿可能需要把连接器插到发动机舱中的某个插槽上,或许不是 USB,但思路应该一样。不难想象另一种情景:开车经过经销商的门店即可升级,或者利用智能手机的无线连接也可以升级。当然也不难想象,以开车经过的方式升级也有潜在的问题。

这个例子也提醒我们,计算机是许多关键系统的核心,并且软件控制着这些系统。MRI(核磁共振)和 CT(电脑断层)扫描等医学成像系统,就是用计算机来控制信号,并生成供医生解读的图像(胶片已经被数字图像取代)。现代汽车都有数十个小型计算机,分别负责管理制动和稳定性控制系统,无论哪个出问题,后果都不堪设想。火车、轮船、飞机也概莫能外。最近,一位飞行员朋友跟我说,他的飞机升级了软件系统,自动驾驶仪的操控方式完全变了一个样。航空交通管制系统、辅助导航设备、电网和电话系统也同样如此。基于计算机的投票器曾经有过严重的缺陷。军事系统更是完全依赖于计算机,而全球的金融系统不也一样嘛?“网络战争”指的是对诸如此类的计算机系统的攻击,已经成为一个流行的名词。这些威胁在现实中存在吗?应该说是存在的。例如,2010 年底“超级工厂”(Stuxnet worm)蠕虫病毒攻击了伊朗核电站的铀浓缩离心机,显然不像是偶然事件。

事实表明,只要软件不可靠不耐用,我们就一定会遇到麻烦。而随着人们对软件越来越依赖,潜在的麻烦也只会越来越大。后面我们还会介绍到,很难写出一点问题都没有的软件。逻辑或实现上的任何一点错误或疏忽,都可能导致程序出问题。即使正常使用中不会发生这些问题,也会给敌人留下可乘之机。

第 4 章

算　　法

什么是软件？一个通俗的比喻是做菜用的菜谱。菜谱会列出做某个菜所需的原材料、烹饪步骤以及预期结果。类似地，程序也要描述待操作的数据，讲清楚要对数据做什么，以及产出什么结果。不过，菜谱与任何程序都不能比，因为它含糊，容易产生歧义。所以这个比喻并不是非常恰当。比如，我家那本《烹饪的快乐》（*Joy of Cooking*）在说到打鸡蛋时，说要“放在一个小碗里：1 个鸡蛋”，但没说必须先把蛋磕开，把壳去掉。

用纳税申报表来作比喻更准确一些：这些表格极其详尽地说明了你应该做什么（“从第 29 行减去第 30 行。如果结果是 0 或更小，则输入 0。给第 31 行乘上 25%，……”）。虽然这个比喻也不完美，但与菜谱相比，纳税申报表在说明计算过程方面更胜一筹：数学计算必不可少，数据从一个位置被复制到另一个位置，后续的计算取决于之前计算的结果。

对于纳税来说，这个过程应该是完整的，无论什么情况下都应该得出一个结果，即应纳税额。应该是毫无疑义的，只要开始的数据相同，任何人都应该得到相同的最终结果。而且应该在有限的时间内完成。从我个人经验来看，这几条都是理想化的，因为术语并不总是很明了，计算说明也比税务机关的说法含糊很多，而且要使用什么数据经常也不好确定。

算法，就是保证特定计算过程正确执行的一系列步骤，它是计算机科学中的菜谱或纳税申报表，只不过编制得更仔细、更准确、更清楚。算法的每一步都表达为一种基本

操作，其含义都是完全确定的，如“两个数相加”。任何事物都没有歧义，输入数据的性质也是既定的。所有可能的情况都会涵盖，而算法绝不会遇到一种它不知道接下来该做什么的情况。（计算机科学家有时候也不免书生气，因此通常会给算法多加一个限定条件：任何算法最终必须停止。根据这个标准，经典的洗发水使用说明“起泡、冲洗、重复”就不能说是算法了。）

设计、分析和实现高效的算法是学院派计算机科学的工作核心，而在现实世界中也有很多算法意义重大。我没有打算滴水不漏地解释或说明各种算法，但我想让大家了解相关的思想，即详尽地描述一系列操作步骤，不管执行这些步骤的实体有没有智能或创造力，都能对这些步骤是什么意思以及如何执行做到毫无疑义。另外，我还想谈一谈算法的效率，也就是计算时间与要处理的数据量之间存在什么关系。为此，我会分析几个常见且容易理解的基本算法。

虽然不必把本章中的每一句话或者偶尔出现的公式都搞明白，但其中的思想还是值得读者深入研究的。

4.1 线性算法

假设我们想找出谁是房间里个子最高的人。我们可以四下里看看，然后猜一猜会是谁。然而，算法则必须精确地列出每一个步骤，从而让不会说话的计算机都能遵照执行。最基本的做法就是依次询问每个人的身高，并记住到目前为止谁最高。于是，我们可能会问“约翰，你多高？玛丽，你呢？”等等。如果我们第一个问的是约翰，那么当时他是最高的。如果玛丽更高，则现在她是最高的人，否则，约翰仍然最高。无论如何，我们都会接着问第三个人。在问完每个人之后，我们就会知道究竟谁最高以及到底有多高。类似的方法还可以找出最有钱的人，或者名字在字母表中最靠前的人，或者谁的生日最接近年底，谁在 5 月出生，以及谁叫克里斯。

会遇到一些复杂的情况。比如如何处理重复的数据，或者说要是有两三个人的身高一样怎么办？我们可以决定只记录第一个人、只记录最后一个人，或者随机记录其中某一个人，再或者记录他们所有人。请注意，找出同样身高的所有人是比较困难的。因

为必须记住所有这些人的名字，不问完最后一个人，我们是无法知道这些信息的。这个例子涉及数据结构，即如何表示计算过程中所需的信息。数据结构对很多算法而言都是非常重要的，但在这里我们不会谈太多。

怎么计算所有人的平均身高？可以询问每个人的身高，每问完一个就加一个（或许可以使用玩具程序来进行累计），最后用累计和除以人数。假设一张纸上写着 N 个人的身高，那这个例子更像“算法”的表达方式如下：

```
把累计和 sum 设置为 0
对这张纸上的每一个身高值 height
  把 height 加到 sum 上
平均身高等于 sum/N
```

但是，如果让计算机来做这件事，就必须多加小心。例如，要考虑到假如纸上没有身高值怎么办？这对人来说不是问题，因为我们知道这意味着什么也不用做。但对计算机来说，我们必须告诉它如何测试这种情况，出现这种情况该怎么办。假如不事先测试，那它就会尝试用零去除 sum，而这个操作是未定义的。算法和计算机必须处理所有可能的情况。如果你看到过“0 美元 00 美分”的支票，或者收到过尚欠余额为 0 元的账单，那就是因为计算机系统没有全面测试所有可能的情况。

如果我们事先不知道有多少个数据项怎么办（这种情况很常见）？那就得重写算法，让它在累计和的同时计算有多少项。

```
把累计和 sum 设置为 0
把项数 N 设置为 0
只要有剩下的身高值要处理
  把下一个 height 加到 sum 上
  给 N 加 1
如果 N 大于 0
  平均身高是 sum/N
否则
  就说没有给出身高值
```

这是避免出现除数为零的问题一种方式，即明确地测试极端情况。

算法的一个关键属性是其效率有多高——对于给定的数据量，它们的处理速度是快还是慢，要花多长时间？对于上面给出的例子，计算机要执行多少步，或者需要花的时间有多长，与它必须处理的数据量成正比：如果房间里的人多出一倍，就要多花一倍时间才能找到最高的人，或者才能计算出平均身高；如果人数是现在的十倍，就要花

十倍的时间。如果计算时间与数据量成正比或叫线性比例，那该算法就叫做线性时间算法或线性算法。以数据量为横坐标，以时间为纵坐标画一条线，得到的将是一条向右上方延伸的直线。我们平时遇到的大多数算法都是线性的，因为它们对某些数据所执行的基本操作是相同的，数据越多工作量也会同比例增加。

线性算法的基本形式都一样。可能需要进行一些初始化，如把累计和的初值设置为 0，或者把最大的身高值设置为一个较小的值。然后依次检查每一项，对它完成一次简单的计算，如计数、与上一个值比较，或进行简单的变换。最后，可能需要再做一些计算，如计算平均值、打印累计和或最大的身高值。如果对每一项执行操作所花的时间相同，那么总时间与数据项数就是呈正比的关系。

4.2 二分搜索

那我们还可以做得更好一些吗？假设我们面前有一大堆打印出来的人名和电话号码，或者一沓名片。如果名字并没有特定的顺序，而我们想找到迈克・史密斯（Mike Smith）的号码，那就必须一个名字一个名字地找，直至找到他的名字为止（或者没找到，因为根本就没有这个人）。如果名字是以字母顺序排列的，我们就可以做得更好。

想想我们是怎么从老式的电话簿中查人名的。首先，我们会从接近中间的地方开始查。如果要找的名字比中间页上的名字在字母表中靠前，那后半本就不用看了，直接翻到前半本的中间（整本电话簿的四分之一处）；否则，前半本就不用看了，直接翻到后半本的中间（整本电话簿的四分之三处）。由于名字按字母顺序排列，每一步我们都知道接下来到哪一半里去找。最终，我们一定会找到那个名字，或者可以断定电话簿里根本就没有这个人。

这个搜索算法被称为二分搜索，因为每次检查或比较都会把数据项一分为二，而其中一半今后就不会再理会了。这其实也是常见的分而治之策略的一个应用。它的速度有多快？每一步都会舍弃一半数据项，因此所需要的步数就等于在处理最后一项之前，最初的项数被 2 除开的次数。

假设最初有 1024 个名字（这个数容易计算）。一次比较，就可以舍弃 512 个。再比较

一次，还剩 256 个，然后是 128 个、64 个、32 个，接着是 16 个、8 个、4 个、2 个，最后剩下 1 个。总共比较了 10 次。显然，2^{10} 等于 1024 并非巧合。比较次数作为 2 的指数就能得到最初的数，而从 1 到 2 到 4……到 1024，每次都是乘以 2。

如果你还记得学校里讲过的对数（没有多少人会记得——谁还记得？），那你应该知道一个数的对数就是底数（这里是 2）要得到该数需要自乘的次数。1024（以 2 为底）的对数等于 10，就是因为 2^{10} 等于 1024。对于我们而言，这里的对数就是要把一个数变成 1，反复除以 2 的次数；或者让 2 反复自乘，得到那个数所需的次数。本书不需要考虑精度或者小数，近似的数字和整数值足矣，纯粹是一种简化。

二分搜索的关键是数据量的增长只会带来工作量的微小增长。如果有 1000 个名字按字母顺序排列，那为了找到其中一个必须检查 10 个名字。如果有 2000 个名字，也只要检查 11 个名字，因为看完第一个名字立即就能舍弃 2000 个中的 1000 个，而这又回到了从 1000 个中查找的情形（检查 10 次）。如果有 1 000 000 个名字，也就是 1000 的 1000 倍，那么前 10 次测试就能减少到 1000，另外 10 次测试即可减少到 1，总共 20 次测试。1 000 000 是 10^6，约等于 2^{20}，因此 1 000 000（以 2 为底）的对数约等于 20。

由此，你就可以知道，在包含 10 亿个名字的名录（全地球的电话簿）中找 1 个名字，也只需要 30 次比较，因为 10 亿约等于 2^{30}。这就是为什么我们说数据量的增长只会带来工作量的微小增长——数据量增长到 1000 倍，只需要多比较 10 次。

我们来验证一下，假设我要从一本旧的哈佛通讯录中找到我的朋友 Harry Lewis，这本 224 页的通讯录中有大约 20 000 个人名。我首先翻到 112 页，看到了 Lawrence。Lewis 在它后面，所以我又翻到 168 页，即 112 页和 224 页中间，找到了 Rivera。Lewis 在它前面，于是我翻到 140 页（112 页和 168 页中间），看到了 Morita。再向前翻到 126 页（112 页和 140 页中间）找到 Mark。随后是 119 页（Little）、115 页（Leitner）、117 页（Li），最后翻到 116 页。这一页大约有 90 个名字，在同一页上又经过 7 次比较，我从几十位 Lewis 中找到了 Harry。这个实验总共比较了 14 次，跟我们预期差不多，因为 20 000 介于 2^{14}（16 384）和 2^{15}（32 768）之间。

分而治之在现实当中也广泛应用于很多比赛的淘汰赛。比赛开始一般都有很多的选手，比如温布尔登网球公开赛男子单打比赛一开始有 128 位选手，每轮比赛都会淘汰

一半，最后一轮剩下两个人，决出一名冠军。这并非巧合，128 是 2 的幂（2^7），所以温布尔登网球公开赛要打七轮。甚至可以想象举办一次全球规模的淘汰赛，即便有 70 亿个参赛者，也只要 33 轮就可以决出冠军。（如果你还记得第 2 章讨论过的 2 和 10 的幂，通过心算也很容易验证这一点）。

4.3　排序

不过，得先把这些名字按照字母顺序排列起来呀，怎么做到呢？如果没有这个先行步骤，就不能使用二分搜索。这就引出了另一种基本算法——排序，把数据按顺序排好，后续搜索才能更快。

假设我们要把一些名字按照字母顺序排好，以便后面更有效地使用二分搜索。那么可以使用一个叫**选择排序**的算法，因为它会不断从未经排序的名字中选择下一个名字。这个算法的技巧，就是前面讨论的找出房间里最高的那个人所用的技巧。

好，让我们就把下面这 16 个熟悉的名字按字母顺序排列一下：

Intel Facebook Zillow Yahoo Picasa Twitter Verizon Bing
Apple Google Microsoft Sony PayPal Skype IBM Ebay

首先是 Intel，它是到目前为止按字母排序后的第一个名字。Facebook 在字母表中更靠前，所以它暂时又成为第一个。Zillow 不靠前，而且直到 Bing 才取代 Facebook，但 Bing 随后又被 Apple 取代。我们接着比对其余名字，没发现一个位于 Apple 之前的，因此 Apple 是这个序列中真正的第一个，当前的结果如下：

Apple
Intel Facebook Zillow Yahoo Picasa Twitter Verizon Bing
Google Microsoft Sony PayPal Skype IBM Ebay

现在，重复上述过程，找到第二个名字，从 Intel 开始——Intel 是未排序名字中的第一个名字。同样，Facebook 取而代之，然后是 Bing 成为第一个元素。第二遍之后的结果如下：

Apple Bing

Intel Facebook Zillow Yahoo Picasa Twitter Verizon

Google Microsoft Sony PayPal Skype IBM Ebay

最终，通过这个算法得到了完全排好的名字列表。

选择排序的工作量有多大？它每次都会遍历剩余的数据项，每次都会找到字母顺序中的下一个名字。对于 16 个名字的排序，查找第一个名字要检查 16 个名字，查找第二个名字需要 15 步，查找第三个名字需要 14 步，依此类推，加起来总共要检查 16+15+14+⋯+3+2+1 个名字。当然，我们也可能很幸运，发现这些名字已经都按字母排好序了。但研究算法的计算机科学家可都是悲观主义者，他们假设的是最坏的情况（即这些名字都是按照字母顺序的反序排列的）。

检查名字的遍数与最初的数据项数成正比（我们例子中的数据项有 16 个，可以用一般化的 N 表示）。而每一遍要处理的项数都比前一遍少一项，所以选择排序算法一般化的工作量是：

$$N+(N-1)+(N-2)+(N-3)+\cdots+2+1$$

这个序列加起来等于 $N\times(N+1)/2$（最简单的办法是把两头的项成对地加起来），也就是 $N^2/2+N/2$。忽略除数 2，可见选择排序的工作量与 N^2+N 成正比。随着 N 不断增大，N^2 最终会大得让 N 也可以忽略不计（例如，如果 N 是 1000，则 N^2 就是 1 000 000）。因此，结果就是工作量近似地与 N^2 即 N 的平方成正比，而这个增长率叫做二次增长。二次增长不如线性增长，事实上，差得很远。要排序的数据项增加到原来的 2 倍，时间会增加到原来的 4 倍；数据项增加到 10 倍，时间会增加到 100 倍；数据项增加到 1000 倍，时间会增加到 1 000 000 倍！这可不太好。

幸运的是，有办法让排序更快一些。我们简单地介绍一种巧妙的方法——快速排序（Quicksort），这个算法是英国计算机科学家托尼·霍尔在 1962 年前后发明的（霍尔获得了 1980 年的图灵奖，获奖理由是包括快速排序在内的多项贡献）。快速排序也是分而治之的一个绝佳示例。

同样，下面还是那些未经排序的名字：

Intel Facebook Zillow Yahoo Picasa Twitter Verizon Bing

Apple Google Microsoft Sony PayPal Skype IBM Ebay

要使用快速排序算法给这些名字排序，首先要遍历一次所有名字，把介于 A 到 M 之间的名字放到一组里，把介于 N 到 Z 之间的名字放到另一组里。这样就把所有名字分成了两个组，每个组里包含一半名字（假设这些名字的分布不会很不均匀）。在我们的例子中，这两个组分别包含 8 个名字：

Intel Facebook Bing Apple Google Microsoft IBM Ebay

Zillow Yahoo Picasa Twitter Verizon Sony PayPal Skype

现在，遍历 A-M 组，把 A 到 F 分成一组，G 到 M 分成另一组；遍历 N-Z 组，把 N-S 分成一组，T-Z 分成一组。到现在为止，遍历了所有名字两次，分成了四个组，每个组包含四分之一的名字：

Facebook Bing Apple Ebay

Intel Google Microsoft IBM

Picasa Sony PayPal Skype

Zillow Yahoo Twitter Verizon

接下来再遍历每个组，把 A-F 分为 ABC 和 DEF，把 G-M 分成 GHIJ 和 KLM；同样，对 N-S 和 T-Z 也如法炮制。这样，就有了 8 个组，每组差不多有 2 个名字：

Bing Apple

Facebook Ebay

Intel Google IBM

Microsoft

Picasa PayPal

Sony Skype

Twitter Verizon

Zillow Yahoo

当然，到最后我们不仅仅要看名字的第一个字母，比如要把 IBM 排到 Intel 前面，把 Skype 排到 Sony 前面，就得继续比较第二个字母。但就这样多排一两遍，即可以得到

16个组，每组 1 个名字，而且所有名字都按字母顺序排好了。

整个过程的工作量有多大？每一遍排序都要检查 16 个名字。假设每次分割都很完美，则每一遍分成的组分别会包含 8、4、2、1 个名字。而遍数就是 16 反复除以 2 直到等于 1 为止除过的次数。结果就是以 2 为底 16 的对数，也就是 4。因此，排序 16 个名字的工作量就是 $16\log_2 16$。在遍历 4 遍数据的情况下，快速排序总共需要 64 次操作，而选择排序则需要 136 次。

该算法可以对任何数据进行排序，但只有在每次都能把数据项分割成大小相等的组时，它才是最有效的。对于真实的数据，快速排序必须猜测数据的中位值，以便每次都能分割出相同大小的组。好在，只要对少量数据项进行采样，就可以估计出这个值。一般来说（忽略某些细节上的差别），快速排序在对 N 个数据项排序时，要执行 $N\log N$ 次操作，即工作量与 $N\log N$ 成正比。这与线性增长比要差一些，但还不算太坏，在 N 特别大的情况下，它比二次增长即 N^2 增长可以好太多了。

为此我做了个实验，随机生成了 1000 万个 9 位数，用于模拟美国的社会保障号，记录了不同规模的分组排序下所花的时间，测试了选择排序（N^2 即二次增长）和快速排序（$N\log N$）。下表中的短划线表示没有做该项测试。

数值个数（N）	选择排序时间（秒）	快速排序时间（秒）
1 000	0.047	—
10 000	4.15	0.025
100 000	771	0.23
1 000 000	—	3.07
10 000 000	—	39.9

精确测量运行时间很短的程序并不容易，因此这些测试数据的误差可能比较大。但无论如何，还是可以（粗略地）看到快速排序意料之中的 $N\log N$ 式的时间增长，同时也能看到尽管选择排序的效率难以与之匹敌，但在 10 000 个数据项以内时还是可以接受的；而且，在每个数量级上，选择排序都被快速排序毫无悬念地抛在了后头。

另外也要注意，在对 100 000 个数值进行排序时，选择排序的时间是对 10 000 的数值进行排序时的近 200 倍，而不是我们期望的 100 倍。几乎可以肯定，这是缓存效应——由于这些数据并没有全部保存在缓存中，所以排序变慢了。

4.4　难题与复杂性

刚才，我们对算法的“复杂性”或运行时间进行了简单的剖析。一个极端是 $\log N$，即二分搜索的复杂性，它表示随着数据量的增加，工作量的增长非常缓慢。最常见的情况是线性增长，或者说简单的 N，此时工作量与数据量是成正比的。然后是快速排序的 $N \log N$，比 N 差（增长快），但在 N 非常大的情况下仍然特别实用。还有就是 N^2，或者二次增长，增长速度太快了，既让人无法忍受又不切实际。

除了这些之外，还有其他很多种复杂性，有的容易理解（例如三次增长，即 N^3，比二次增长还差，但道理相同），有的则很难懂，只有少数专业人士才会研究。但有一个还是非常值得了解一下，因为它在现实当中很常见，而且从复杂性上说特别糟糕，这就是所谓的指数级增长，用数学方法表示是 2^N（与 N^2 可不一样）。指数级算法的工作量增长极快：增加一个数据项，工作量就会翻一番。从某种意义上讲，指数级算法与 $\log N$ 算法是两个极端，后者数据项翻一番，工作量才增加一步。

什么情况下会用到指数级算法呢？那就是除了一个一个地尝试所有可能性，没有更好的办法的情况。谢天谢地，指数级算法总算是有点用武之地的。有些算法，特别是密码学中的算法，都是让特定计算任务具有指数级难度的。对于这样的算法，只要选择了足够大的 N，其他人在不知道某个秘密捷径的情况下，是不可能通过计算直接解决问题的。第 10 章还会再介绍密码学。

现在你只要知道有些问题容易解决，而有些问题则要难得多就可以了。实际上，关于解决问题的难易程度，也可以表达得更加精确一些。所谓“容易”的问题，都具有“多项式”级复杂性。换句话说，解决这些问题的时间可以用 N^2 这样的多项式来表示，其中指数可以大于 2，但都是可能被解决的。（忘了什么是多项式啦？不要紧，多项式在这里就是指一个变量的整数次幂，比如 N^2 或 N^3。）计算机科学家称这类问题为“P”（即“Polynomial”，多项式），因为它们可在多项式时间内解决。

现实中大量的问题或者说很多实际的问题似乎都需要指数级算法来解决，也就是说，我们还不知道对这类问题有没有多项式算法。这类问题被称为“NP”问题。NP 问题的特点是，它可以快速验证某个解决方案是否正确，但要想迅速找到一个解决方案却很难。NP 的意思是“非确定性多项式”（nondeterministic polynomial），这个术语大概

的意思是：这些问题可以用一个算法在多项式时间内靠猜测来解决，而且该算法必须每次都能猜中。在现实生活中，没有什么能幸运到始终都做出正确的选择，所以这只是理论上的一种设想而已。

很多 NP 问题都会牵扯大量技术细节，三言两语也解释不清楚。不过倒是有一个问题很好解释，乍一看还挺有意思的，而且其实际应用也比较广。

这就是“旅行推销员问题”（Traveling Salesman Problem）。一个推销员必须从他居住的城市出发，到其他几个城市去推销，然后再回家。目标是每个城市只到一次（不能重复），而且走过的总距离最短。这个问题跟最短校车或者垃圾车路线有异曲同工之妙。很早以前我在研究这个问题的时候，其原理经常被应用于设计电路板上孔洞的位置，或者部署船只到墨西哥湾的特定地点采集水样。

旅行推销员问题已经被仔细推敲了 50 多年，尽管能用它来解决的问题更加多样化了，但解决方案的核心依然是从所有路径中更巧妙地找出最短路径。同样，有许多的其他问题，尽管类型不同，形式各异，也都面临同样的命运：我们没有什么好办法有效地解决它们。

对于研究算法的人来说，这个现实令人沮丧。我们不知道到底是这些问题本质上就很难解决呢，还是因为人类不够聪明，所以至今都没有找到更好的解决办法。当然，不管怎么说，人们更愿意相信它们“本质上就很难解决”。

1970 年，斯蒂芬 · 库克证明了一个非同小可的数学结论，就是说所有这些问题其实都是等价的，只要我们找到一个多项式时间算法（复杂性类似 N^2）解决其中一个问题，那我们据此就能找到所有问题的多项式时间算法。库克因此获得了 1982 年的图灵奖。

美国克雷数学研究所（Clay Mathematics Institute）公布了 7 个悬而未决的问题，解决其中一个就可以获得 100 万美元奖金。而问题之一是：P 是否等于 NP？换句话说，这些难题到底跟那些简单的问题是不是一类？（7 个问题中的另一个，可以追溯到 20 世纪初的“庞加莱猜想”，已经被俄罗斯数学家格里戈里·佩雷尔曼解决，奖金已经在 2010 年发放。所以，现在还剩下 6 个待解决的问题——为了防止别人捷足先登，解题要趁早噢 ~ ）

对于这种复杂性，有几个地方需要特别注意。虽然 P=NP 问题很重要，但它更多的是一个理论问题，而不是一个实际问题。正如计算机科学家所说的，复杂性结果就是“最坏的”结果，有些问题的实例可能需要投入全部时间和精力，但并不是所有实例都那么难解决。这些问题也具有“渐近”的特点，也就是说，只有 N 值特别大的情况下才值得考虑。在现实生活中，或许大多数问题都能找到简单的解决办法，或许从实用角度看，近似的结果也是完全可以接受的，或许 N 很小，考虑不考虑渐近问题根本无关紧要。举例来说，如果你只需要对几十或者几百个数据项进行排序，那选择排序可能就足够快了，尽管其复杂性是二次方的，而且与快速排序的 $N \log N$ 相比是每况愈下。如果你只需造访五六个城市，要尝试所有可能的路线不是什么难事儿，但如果是 60 个城市，就有点不可行了，而 600 个城市也这么做根本就是不可能的。最后，在大多数情况下，一个近似的解决方案可能就足够好了，完全没有必要追求所谓的绝对最佳方案。而能够给出合理近似答案的算法可能要多少有多少，其中很多可能还更切合实际。

另一方面，一些重要的应用，如加密系统，则完全是建立在某个特定的问题确实极难解决的基础之上的。因此，你若能发现出一种攻击方法，无论其有多么不切实际，也都是意义非凡的。

4.5　小结

重塑人们对“我们能计算多快”的认识，多年来一直是计算机科学研究的主题。而用数据量来表示运行时间/次数（如 N 、$\log N$ 、N^2或 $N \log N$），则是这一领域研究成果的集中体现。它不去纠结于这台计算机是不是比那一台更快，或者你是不是一个比我更优秀的程序员之类的问题，而是抓住了程序或算法背后的复杂性。正因为如此，才非常合适比较或推断出某些计算是否可行。（一个问题固有的复杂性和解决这个问题的算法的复杂性并不是一个概念。比如，排序是一个 $N \log N$问题，但快速排序是一个 $N \log N$ 算法，而选择排序则是一个 N^2算法。）

算法和复杂性的研究是计算机科学的一个重要组成部分，既有理论也有实践。我们感兴趣的是哪些问题可以计算，哪些不可以，以及如何在无需更多内存的情况下计算得更快（或者是同样速度下使用更少的内存）。我们期待全新的、更好的计算方法。快

速排序就是一个典型的例子，尽管它已经出现很多年了。

现实生活中，有许多重要的算法比我们这里介绍的简单的搜索和排序更专业更复杂。例如，压缩算法旨在让声音（MP3）、图片（JPEG）和电影（MPEG）占用更少的存储空间。错误检测和校正算法也很重要。数据在存储和传输（例如通过嘈杂的无线信道）过程中可能会受到损害；控制数据冗余的算法可以检测甚至纠正某些错误。在第 9 章介绍通信网络的时候，我们还会继续讨论这个话题。

密码学高度依赖于算法，它需要发送只让好人看懂而不能让坏人破解的加密消息。我们将在第 10 章讨论密码学，因为它与通过计算机交换私密信息紧密相关。

对于必应和谷歌等搜索引擎而言，算法同样至关重要。从原理上讲，搜索引擎所做的大量工作都很简单，无非是收集网页、组织信息，使其便于搜索，所不同之处在于数据规模极大。如果每天有数十亿次查询，要搜索数十亿个网页，那么即使 $N \log N$ 的复杂性也是无法接受的。为了跟上日益增长的 Web 数据量，满足我们通过它进行搜索的需求，人们在改进算法和编程方面投入了大量聪明才智，以确保搜索引擎能够足够快。我们将在第 11 章里更详细地讨论搜索引擎。

第 5 章

编程与编程语言

上一章我们讨论了算法。算法是忽略具体实例而对过程进行的一种抽象或理想化的描述，是分毫不差且没有歧义的“菜谱”。算法通过一组确定的基本操作来表达，这些操作的含义是完全已知且明确的。算法描述了应用这些基本操作的一系列步骤，涵盖所有可能的情况，而且保证最终能够停止。

另一方面，**程序**则不是抽象的，它陈述了一台真正的计算机要完成某个任务所必须执行的具体步骤。程序之于算法，犹如建筑之于图纸，一个是实际存在的，一个是理想化的。

换一个角度看，程序又是以计算机能够直接处理的某种形式表达出的一个或多个算法。程序必须考虑实际的问题，比如内存不足、处理器速度不快、无效或恶意的输入、网络连接中断，以及（看不见摸不着，但却经常会导致其他问题恶化的）人性弱点。因此，如果说算法是理想化的菜谱，那程序就是让烹饪机器人冒着敌人的炮火为军队准备一个月的给养所需的操作说明书。

好啦，打比方只能到此为止了。本章我想给大家好好地讲一讲什么是编程，让你理解编程是怎么回事儿。当然，光凭这些要想成为专业的程序员还是远远不够的。编程不是件容易的事儿，很多细节都必须准确无误，而稍有差池就可能铸成大错。

我们需要或者希望计算机无所不能，如此一来就需要巨大的编程工作量，而世界上却没有那么多程序员。因此，让计算机代替人处理更多的编程细节就成为这个领域永恒的话题。这自然也就引出了编程语言，即让我们能够表达完成某个任务所需计算步骤的语言。

同样，管理一台计算机的资源也十分困难，而现代计算机的复杂性更是让这种困难有增无减。因此，我们也需要让计算机来控制自己的操作，而由此就有了所谓的操作系统。编程和编程语言是本章的主题，而软件系统，特别是操作系统将会在下一章讨论。

读者可以不去关注本章代码示例的语法细节，但比较一下它们在表达计算步骤时的异同还是很有意思的。

5.1 汇编语言

1949 年，EDSAC 的诞生标志着第一批真正可编程的电子计算机登上了历史舞台。那时候，给这些计算机编程要把表示指令和数据的数值打在穿孔卡片或纸上，然后把这些数值加载到存储器中执行。用这种编程方式哪怕写一个非常小的程序也十分艰难。首先是不可能一次就做对，其次是发现错误以后修改、增删指令或数据也不容易。

20 世纪 50 年代，出现了能代替人处理某些琐事的程序，因而程序员可以使用有意义的单词来表示指令（如用 `ADD` 代替数字 `5`），使用名字来指代特定的内存位置（如用 `Sum` 代替数字 `14`）。这其中蕴含的用程序操作程序的思想，一直都是软件领域各种重大进步的核心驱动力。这种代替人执行具体操作的程序被称为**汇编器**（assembler），因为它最初也用来组装（assemble）程序中由其他程序员事先写好的部分。相应的语言叫做汇编语言，而这个层次上的编程叫做**汇编语言编程**。第 3 章给玩具计算机写程序所用的语言就是一种汇编语言。有了汇编器，给程序添加或删除指令就方便多了，因为汇编器会负责跟踪数据和指令在存储器中的位置，程序员就不必管这些琐碎的事儿了。

不同处理器的汇编语言只能用于为该处理器编写程序。汇编语言通常都与 CPU 的指令一一对应，能够以特定方式将指令编码为二进制格式，也知道信息在存储器中如何存放。这也就意味着，用某种 CPU（如 Mac 或 PC 中的 Intel 处理器）的汇编语言编写的程序，与在不同 CPU（如手机中的 ARM 处理器）上完成相同任务的其他汇编程序差别会很大。把为一种处理器编写的汇编程序移植到其他处理器，实际上接近于重写一遍执行相同任务的程序。

举个例子，我们的玩具计算机使用了三条指令把两个数加起来，并将结果保存到存储器中的一个位置上：

```
LOAD X
ADD Y
STORE Z
```

这个过程在当前各种处理器上都是类似的。但是，在带有另一种不同指令表的 CPU 上，上述计算可能只需要一条指令：

```
ADD X, Y, Z
```

为了让这个玩具程序能在其他计算机上运行，程序员必须熟悉两种处理器，并小心翼翼地把它们从一个指令集转换到另一个指令集。这活儿可不好干。

5.2 高级语言

20 世纪 50 年代末到 60 年代初，计算机在代替程序员做更多事方面又前进了一大步，而这无疑也是人类编程史上最重要的一步。那就是独立于任何 CPU 体系结构的“高级”编程语言问世了。高级语言让人类得以用接近自然语言的方式来表达计算过程。

用高级语言编写的代码经过一个翻译程序，可被翻译为目标处理器的汇编指令。这些汇编指令则会进一步被转换为比特，从而能够加载到存储器中并执行。这个翻译程序通常被称作**编译器**——同样是一个信息量有限的老术语。

上面计算 X、Y 两个数之和并把结果保存在 Z 中的例子，在大多数高级语言中都可以写成这样：

```
Z = X + Y
```

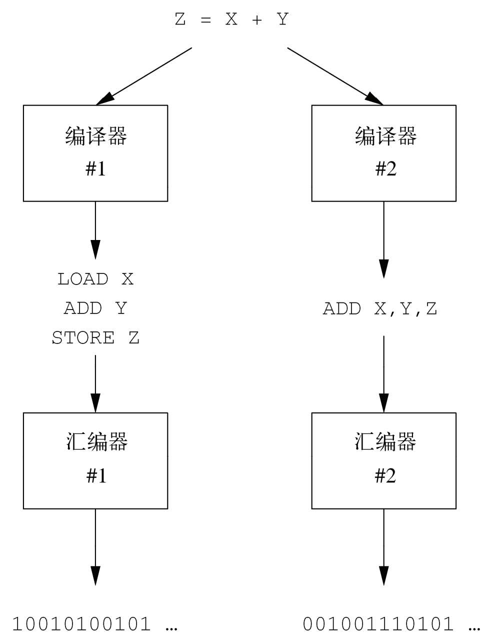
这行代码的意思是：从存储器中的位置 X 和 Y 中取得值，把它们加起来，然后把结果保存到存储器中的位置 Z。

一个编译器可能把这个玩具程序转换成三条指令，另一个编译器则可能把它转换为一条指令。而相应的汇编器将负责把各自的汇编语言指令转换为实际的位模式，同时为 X、Y、Z 这几个量在存储器中留出位置。当然啦，针对这两台计算机的位模式也不一样。

上述过程如下图所示，它显示了相同的输入表达式，通过两个不同的编译器和它们各自的汇编器，产生了不同的指令序列。

在实际当中，编译器在内部可能会被分成一个“前端”和多个“后端”。“前端”负责把高级语言的程序转换为中间形式，而“后端”则负责把中间表现形式转换成不同体系结构的汇编指令。这种做法要比使用多个完全不同的编译器更简单。

相比汇编语言，高级语言拥有很多优势。首先，它让更多的人得以学会编程，而且编程效率也大大提高。用高级语言编程接近人类的思维方式，因此学习和使用的难度都降低了。人们不需要熟悉 CPU 指令表，就可以使用高级语言高效地编程。其次，高级语言程序独立于各种体系结构，通常无需任何修改即可在不同的体系结构上运行，只要像上图所示换个编译器编译一下就行。于是，程序可以只写一次，随处运行了。这也大幅降低了为多种计算机开发程序的成本。而编译环节也为发现各种拼写错误、语法错误（如少写括号或操作未定义的量）等疏漏提供了机会，在生成可执行程序之前必须纠正这些错误。这些错误在汇编语言程序中很难发现，因为必须假设汇编指令的任何序列都合法。（当然，语法正确的程序仍有可能充斥着各种编译器检测不出来的语义错误。）高级语言的重要意义无论怎么强调都不过分。



接下来我想用五种最重要的高级语言来编写同样的程序，好让大家一窥它们之间的

异同。每个程序完成的操作都与第 3 章中那个玩具程序一样，即把一系列输入值加起来，如果读到的输入值为 0，则打印累计和并停止运行。它们的结构也都相同：命名程序要使用的量，把保存累计和的量初始化为 0，读取数值，如果遇到了 0 则打印累计和。不要太过关注语法，这几个例子主要是为了让你对不同语言写的程序有个大致的印象。

第一批高级语言专注于特定的领域。其中一门最早的语言叫做 FORTRAN，这个名字源自“Formula Translation”（公式转换）。FORTRAN 由约翰・巴库斯（John Backus）在 IBM 领导的一个小组开发，在表达科学和工程计算方面非常成功。许多科学家和工程师（包括我）学习的第一门编程语言就是 FORTRAN。FORTRAN 到今天仍然有很多用户。自 1958 年以来，FORTRAN 经历了几次大的变革，但其核心没有变过。巴库斯 1977 年获得图灵奖，其中部分原因就是他领导开发了 FORTRAN。

下面就是加总一系列数值的 FORTRAN 程序：

```
      integer num, sum
      sum = 0
   10 read(5,*) num
      if (num .eq. 0) goto 20
      sum = sum + num
      goto 10
   20 write(6,*) sum
      stop
      end
```

这是用 FORTRAN 77 写的，如果用它较早的方言或者最新版本的 FORTRAN 2008 来写，多少会有些不同。看着这段程序，恐怕你能想象出来怎么把它转换为玩具汇编语言。

20 世纪 50 年代末的第二个主要的高级语言是 COBOL（Common Business Oriented Language，面向商业的通用语言），格蕾斯・霍普（Grace Hopper）对汇编语言高级替代品的研究对它产生了重大影响。霍普与霍华德・艾肯（Howard Aiken）当时使用的是哈佛 Mark I 和 II（当时的机械计算机），后来又使用过 Univac（Universal Automatic Computer，通用自动计算机）。她是认识到高级语言和编译器具有巨大潜力的先驱之一。COBOL 是专门针对商业数据处理的语言，其功能非常适合表达库存管理、开发票、做工资等方面的计算。COBOL 现在也有人在用，虽然变化较大但仍然有它自己的特点。

BASIC（Beginner's All-purpose Symbolic Instruction Code，初学者通用符号指令代码）也是当时问世的一门语言，是约翰 · 凯梅尼（John Kemeny）和汤姆 · 库尔茨（Tom Kurtz）于 1964 年在达特茅斯开发出来的。BASIC 当初的设计目标是要成为学习编程的入门语言。它特别简单，只需要非常有限的计算资源，因此也成为了第一批个人计算机中的第一个高级语言。事实上，微软公司的创始人比尔 · 盖茨和保罗 · 艾伦发迹，也是始于为 1975 年的 Altair 微型计算机编写 BASIC 编译器，这个编译器是微软公司的第一个产品。今天，Microsoft Visual Basic 作为 BASIC 的一个主要分支，仍然被微软公司积极地维护着。

在计算机价格昂贵、速度又慢而且资源有限的时期，人们都担心用高级语言写出来的程序效率太低，因为编译器生成的汇编代码远不如一个熟练的汇编程序员写得好。编译器作者付出了很大努力，希望生成的代码能够达到手写代码一样的高质量，而这为高级语言的流行奠定了基础。今天，计算机速度提升了上百万倍，而且有了充足的内存，程序员很少需要担心指令级的效率问题，尽管编译器和编译器作者仍然很关心。

FORTRAN、COBOL 和 BASIC 获得成功的部分原因，是它们都专注于某个特定的领域，而且有意避免大而全的定位。20 世纪 70 年代，出现了专门为“系统编程”开发的语言。所谓系统编程，就是编写汇编器、编译器、编程工具乃至操作系统等程序员使用的工具。迄今为止，这些语言中最成功的是 C，由丹尼斯 · 里奇（Dennis Ritchie）于 1973 年在贝尔实验室开发，至今仍然有着非常广泛的应用。从那时到现在，C 的变化很小，今天的一段 C 程序与 30 年前的相比，几乎没有多大差别。

为了便于比较，下面就来看看同样的“加总一系列数值”的 C 程序：

```
#include <stdio.h>
main() {
  int num, sum;
  sum = 0;
  while (scanf("%d", &num) != EOF && num != 0)
     sum += num;
  printf("%d\n", sum);
}
```

20 世纪 80 年代又出现了 C++，是比雅尼 · 斯特劳斯特鲁普（Bjarne Stroustrup）同样在贝尔实验室开发的，定位是应对大型程序开发过程中的复杂性。C++由 C 发展而来，而且看起来也跟 C 相似。多数情况下，C 程序也是有效的 C++程序（上面的程序就是），

但反之却绝对不行。下面这个加总一系列数值的例子是用 C++写的（多种写法中的一种）：

```
#include <iostream>
using namespace std;
main() {
   int num, sum;
   sum = 0;
   while (cin >> num && num != 0)
      sum += num;
   cout << sum << endl;
}
```

今天，我们在计算机中使用的大部分软件都是用 C 或 C++编写的。我写这本书所用的 Mac，其中安装的大多数软件都是用 C 和 Objective-C（C 的一种方言）写的。一开始我用 Word（C 和 C++程序），备份则放在 Unix 和 Linux（都是 C 程序）操作系统上，而我上网使用的是 Firefox 和 Chrome（都是用 C++写的）。

20 世纪 90 年代，随着因特网和万维网的发展，更多语言被开发出来。计算机处理器的速度继续加快，内存容量继续增大，而编程是否高效、是否便捷变得比机器效率更重要，此时诞生的 Java 和 JavaScript 在这方面考虑得比较多。

20 世纪 90 年代初，詹姆斯·高斯林（James Gosling）在 Sun Microsystems 公司开发了 Java。Java 最初的目标是开发小型嵌入式系统，例如家用电器和电子设备中的系统，因此对速度要求不高，但对灵活性的要求很高。Java 的目标后来变成了在网页中运行，虽然没有成功，但它在 Web 服务中的应用却非常广泛：打开 eBay 之类的网站，虽然你的计算机在运行 C++和 JavaScript 程序，但 eBay 可能正在用 Java 来生成网页，然后发给你的浏览器。Java 比 C++简单（但复杂度有越来越接近的趋势），但比 C 复杂。另外，由于去掉了一些危险的特性，并且内建内存管理等避免出错的机制，因此 Java 也比 C 安全。出于这个原因，Java 普遍成为编程课上要学习的第一门语言。

以下是用 Java 写的加总一系列数值的程序：

```
import java.util.*;
class Addup {
    public static void main (String [] args) {
       Scanner keyboard = new Scanner(System.in);
       int num, sum;
       sum = 0;
```

```
            num = keyboard.nextInt();
            while (num != 0) {
                sum += num;
                num = keyboard.nextInt();
            }
            System.out.println(sum);
        }
    }
```

这段程序比其他语言写的稍微长一点，这对 Java 而言不算不正常。不过，通过组合几个计算，我还可以减少两三行代码。

这也引出了程序和编程方面一个非常重要的共识：针对某个特定的任务，总会有多种写程序的方式。从这个意义上说，编程就像是文学创作。没错，风格以及恰如其分地运用语言对写作至关重要，对写程序同样至关重要，而且还是区分真正伟大的程序员与普通程序员的标志。程序员对特定的计算任务可以有如此丰富的表达方式，也意味着不难识别从他人程序中复制的非原创代码。我每次上编程课的时候都会着重强调这个观点，但还是有学生认为改改名字或者挪挪代码，就可以掩盖剽窃的事实。很抱歉，这是行不通的。

JavaScript 同样是 C 衍生语言大家族的一员，但它与 C 的差别也非常大。它是布兰登·艾奇（Brendan Eich）1995 年在网景公司开发的。最初，设计 JavaScript 的意图是在浏览器中实现网页的动态效果，而今天，几乎每个网页里或多或少都会包含一些 JavaScript 代码。关于 JavaScript，我们在第 7 章还会详细讨论。但为了便于大家比较，下面给出 JavaScript 版的加总一系列数值的程序：

```
var num, sum;
sum = 0;
num = prompt("Enter new value, or 0 to end");
while (num != 0) {
    sum = sum + parseInt(num);
    num = prompt("Enter new value, or 0 to end");
}
alert("Sum = " + sum);
```

从某些方面看，JavaScript 是所有语言中最容易实验的。这门语言本身也简单。你不需要为 JavaScript 程序找编译器，每个浏览器都内置了一个。于是乎，计算结果可以迅速出现在你眼前。后面我们还会介绍，给这个程序添上几行代码，然后把它放到网页中，全世界的任何人就都可以使用它了。

以后的语言将何去何从？我猜想，人们将继续使用更多的计算机资源让编程变得更容易。而且我们还会继续发展那些对程序员来说更安全的语言。比如，C 语言就像一柄双刃剑，用它写出的程序很容易遗留错误，而等到发现的时候却已经为时已晚（或许已经被用到了不法用途上）。使用较新的语言更容易防止或至少能检测到某些错误，但有时代价是运行速度慢或占用内存多。大多数情况下，这种取舍的方向是没错的，然而肯定还是有很多应用（比如汽车、飞机、航天器和武器的控制系统）对代码的效率和速度要求相当高，因此像 C 这样的高效语言仍然会有人用。

虽然所有语言在形式上都是等价的（都可以模拟图灵机），但这绝不是说它们都适用于所有的编程任务。写一个控制复杂网页的 JavaScript 程序，与写一个实现 JavaScript 编译器的 C++程序仍有天壤之别。同时擅长这两种编程任务的程序员并不多见，经验丰富的专业程序员也可能熟悉或粗通十几种语言，但他们不会对多种语言都同样熟练。

现在的编程语言多达几千种甚至上万种，但真正广泛使用的恐怕连 100 种都到不了。为什么会这么多？前面也提到过，每种语言都代表了对效率、表达力、安全性和复杂性的取舍。许多语言显然是为了弥补之前语言的不足才被发明的，它们不仅吸取了之前语言的教训，还能利用更多的计算资源，通常也会受到设计者个人偏好的强烈影响。新的应用领域也会催生专门面向该领域的新语言。

不管怎么样，编程语言都是计算机科学的一个重要而迷人的部分。正如美国语言学家本杰明・沃尔夫（Benjamin Whorf）所说："语言塑造我们的思维方式，决定我们可以思考什么。"这个论断是否适用于自然语言还有争议，但对于我们发明的告诉计算机去做什么的人造语言来说，好像还是挺靠谱的。

5.3 软件开发

现实中的编程往往是大规模的。大规模编程的方法与任何人想写一本书或承担任何大项目时一样：先搞清楚要做什么，然后从大概的规程着手，将其一级一级分解为较小的任务，再分别完成这些小任务，同时保证它们能够组合在一起。在编程中，每个小任务意味着一个人用某种编程语言可以写出来的精确的计算步骤。确保不同的程序员

编写的代码能够在一起工作很有挑战性，而做不到这一点则是错误的主要来源。例如，1999 年美国航空航天局发射的火星气象卫星坠毁，就是因为飞行系统软件在计算推动力时使用的是公制单位，但输入的路线校正数据使用的则是英制单位，使得该卫星太过接近火星表面所致。

前面演示不同语言的示例都没有超过 10 行代码（Java 除外）。类似这种学过编程入门课程就可以写出来的小程序，也可能包含几十甚至几百行代码。我曾经写过的第一个“真正的”程序（所谓真正，指的是供很多人用过），大约有 1000 行 FORTRAN 代码。那是一个简单的智能文字处理器，用于排版和打印我的论文，我毕业后这个程序就由一个学生组织接管，随后又被人使用了差不多五年之久。真是忆往昔峥嵘岁月稠啊！今天，稍有价值的程序可能都会包含几千甚至几万行代码。参加我的编程实践课的学生，分为几个小组，写两三千行代码通常需要 8 到 10 周时间，包括系统设计和学习一两门新语言的时间。他们的作品通常都是一些 Web 服务，涉及学生间的二手书交易或者为访问某些大学数据库提供便利。

编译器或 Web 浏览器可能有几十万到一百万行代码。大型系统则可能有几百万甚至上千万行代码，由数百或数千人共同开发，开发时间也长达几年乃至几十年。这种规模的软件需要程序员、测试人员、文档编写人员协同工作，还要有开发计划、最终期限，层层管理，历经无穷无尽的会议一步步走下去。（据我的一位同事说，他曾参与过一个重要系统的开发，针对该系统的每一行代码都要开一次会。那个系统有几百万行代码，所以他说得可能太夸张了，但老资格的程序员可能还是会说：“这话说得不过分。”）

5.3.1　库、接口和开发工具包

今天如果要盖一间房子，你不必自己伐木取材，烧土做砖，你可以去买各种预制件，比如门、窗、卫浴器具、火炉和热水器。盖房起屋仍然是一个艰巨的任务，但却是你完全能够做到的，因为你可以使用其他人的工作成果，还有各种基础设施做保障。实际上是有一个完整的产业链条，从各个方面为你提供帮助。

编程未尝不是如此。所有重要的程序几乎没有从零开始写的，有许许多多别人已经写出来的东西可以拿来就用。举个例子，如果你在为 Windows 或 Mac 写程序，那么有

很多库都能提供预制的菜单、按钮、文本编辑器、图形、网络连接、数据库访问功能。实际上，你的大部分时间要用来理解这些组件，然后再以自己的方式把它们“粘”在一块儿。当然，这些组件有很多也依赖其他更简单、更基础的库，经常要分好几层。而在最下层，就是支持所有程序运行的操作系统，它是负责管理硬件并确保一切井然有序的程序。下一章我们会讨论操作系统。

在最基本的层次上，编程语言提供了一种机制，叫做函数。这样，程序员就可以写出一段执行某个任务的代码，然后以某种形式把它包装起来，提供给别的程序员在其他程序里使用，而这些程序员不必知道那些代码具体如何完成该任务。比如，前几页中的 C 程序包含以下几行代码：

```
while (scanf("%d", &num) != EOF && num != 0)
   sum += num;
printf("%d\n", sum);
```

这里的代码“调用”（也就是使用）了两个 C 语言内置的函数：scanf 和 printf。其中，scanf 用于从输入源读取数据（类似我们玩具程序中的 GET），而 printf 打印输出（类似 PUT）。函数有函数名，接收完成任务所需的输入数据值，完成计算后把结果返回给调用它的程序。这里的语法及其他细节都是 C 语言特有的，可能会与其他语言不一样，但内在思想是一致的。函数使我们可以基于组件搭建程序，而这些组件则是独立创建、可以由任何程序员按需使用的。把一组相关的函数集合起来，就叫做**库**。例如，C 有一个标准函数库，用于读写磁盘和其他地方的数据，scanf 和 printf 就是这个库里的函数。

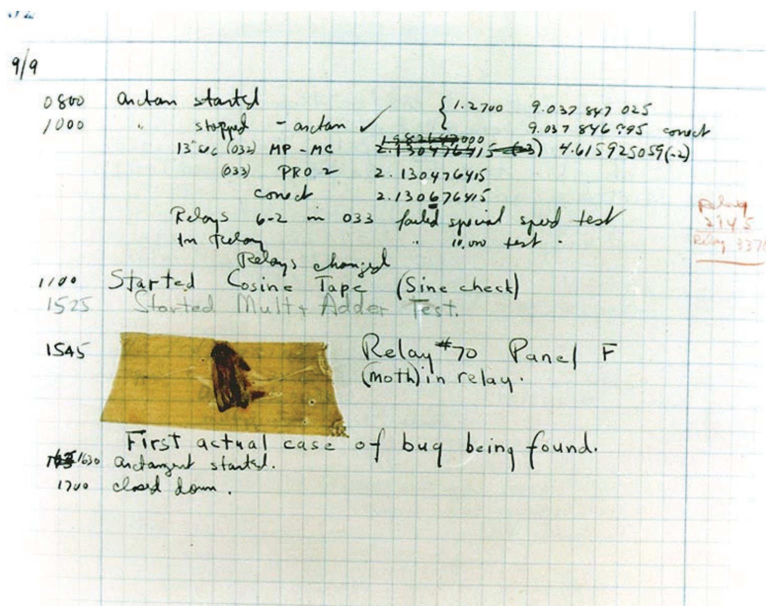
函数库提供的服务是通过 **API**（Application Programming Interface，应用编程接口）的形式描述给程序员的。API 会罗列出所有函数，说明每个函数的用途、用法、需要的输入数据，以及生成什么值。API 也会描述数据结构，也就是传进来传出去的数据的组织形式，以及为请求服务必须遵守哪些条条框框和计算将返回什么结果。这种说明书必须面面俱到、严谨准确，因为基于它编写的程序最终会由一台不会说话的计算机而不是一个随和友善的人去解读。

API 文档中不仅包含对语法的要求，也包括大量辅助说明，用以帮程序员更有效地使用函数库。今天的大型系统开发通常都会用到 **SDK**（Software Development Kit，软件开发工具包），以便程序员在极其复杂的软件库里找到有用的函数。例如，苹果公司

为开发 iPhone 和 iPad 程序提供了编程环境和相应的支持工具，谷歌也为 Android 手机开发提供了类似的 SDK，微软则为使用不同的语言、针对不同的设备编写 Windows 代码提供了各种开发环境。

5.3.2 bug

可惜的是，没有多少程序第一次就能正常运行。我们周围的这个世界太复杂了，而程序也反映了其复杂性。编程要求对细节极端关注，而能做到这一点的人却不多。正因为如此，任何规模的程序都会包含错误，也就是说，它们在某些情况下会做错一些事或者得出错误的答案。这些缺陷被称为 bug，这个词是因我们前面提到的格蕾斯·霍普而流行起来的。那还是 1947 年，霍普的同事在哈佛 Mark II（他们当时使用的一种机械计算机）中发现了一只虫子（死了的蛾子），结果她就说她们是在给这台机器“除虫”（debug）。那只死虫子后来被保存下来，还做成了标本供后人瞻仰。如果你去华盛顿，可以在史密森尼美国历史博物馆里看到它，下面就是它的照片。



然而，用“bug”代指错误并不是霍普的发明，这个用法可以追溯到 1889 年。以下是摘自《牛津英语词典》（第 2 版）的解释：

> bug. 机器、图纸等中的缺陷或错误。起源：美国
> 1889 年《蓓尔美街报》，3 月 11 日，1/1 我听说啊，爱迪生先生连续两个晚上都在找他留声机里的“bug”——其实就是在排除故障，但听起来好像所有问题都是因为有个虫子钻进去引起的。

导致 bug 的原因多种多样，甚至可以写出一本书来专门论述（确实有这类书）。其中比较常见的原因包括忘记处理可能发生的情况、在测试某个条件时写错了逻辑或算术表达式、用错了公式、访问了没有分配给程序的存储器、错误地操作了某种数据、没有验证用户输入等等。

举个例子吧，下面是两个 JavaScript 函数，用于实现摄氏度与华氏度的相互转换：

```
function ctof(c) {
   return 9/5 * c + 32;
}
function ftoc(f) {
   return 5/9 * f - 32;
}
```

其中一个函数有错误。你看出来了吗？一会儿我再告诉你。

在实际的编程中，很大一部分工作是在编写代码的同时测试代码。很多软件公司的测试人员比程序员还要多，目的就是尽可能在把产品交给用户之前发现更多的 bug。查找 bug 不容易，但你至少可以发现那些会经常出现的。怎么测试上面的温度转换函数呢？你肯定会想，用几个知道结果的简单测试用例试一试，比如摄氏 0 度和 100 度，结果应该是华氏 32 度和 212 度。这两个测试都没有问题。

但相反方向的测试，即从华氏温度转换为摄氏温度则没有那么顺利。函数转换的结果是 32 华氏度等于−14.2 摄氏度，而 212 华氏度等于 85.8 摄氏度，错得离谱了。问题在于，在乘 5/9 之前，必须把华氏温度值减 32 用括号括起来。`ftoc` 中正确的表达式应该是：

```
return 5/9 * (f - 32);
```

还好，测试这个函数并不难。不过由此可以想见，要调试一个上百万行的大程序，从中找出并不那么显明的问题，将会是多大的工作量啊。

软件中的错误如果暴露到网络上，就会成为可能遭受攻击的漏洞。攻击者通常会利用这些漏洞以自己的恶意代码重写内存。正是由于 bug 广泛存在，所以各种重要的软件才会频繁升级，比如浏览器，它已经成为很多网络黑客关注的焦点。就拿我使用的 Firefox 来说吧，短短 18 个月就经历了 19 次修订，共修复近 100 个安全漏洞。这可不是什么特例，但也并不意味着 Firefox 程序员都不够格。这说明编写一个耐用的程序非常难，而坏人始终都在寻找你的弱点。

现实中软件面临的另一个复杂性在于外界环境瞬息万变，因此程序必须不断适应新情况。新的硬件问世后，它所需要的软件可能得进行系统级的改动。新的法律法规出台，程序的逻辑可能就必须调整——众所周知，税法每次有什么变化，相关软件都要升一次级。计算机、工具和语言过了时，就需要有新的替代品问世。数据格式过时的情况更加常见——今天的 Word 软件打不开 20 世纪 90 年代初写就的 Word 文档。自然，存储和处理这些数据的物理设备也一样在代代更替。而随着人的退休、死亡或被公司解雇，专业知识也会消逝。学生在校期间开发的系统随着他们的毕业，也面临相同的遭遇。

必须持续不断地小幅更新，这是软件开发和维护的一大问题，但是不做还不行。如果不那么做，程序就会遭遇“比特腐烂”，一段时间之后，也许就不能用了，或者想更新都更新不了了，因为重新编译无法通过，或者它所依赖的库已经变得太多了。与此同时，无论更新还是修复问题，通常都会产生新的 bug，或者改变用户熟悉的行为。

5.4 软件资产

软件所有权引发了很多棘手的法律问题，我认为比硬件的问题还要多，但作为一个程序员，这也可能是我的偏见。与硬件相比，软件是一个比较新的领域，1950 年之前，还没有软件呢。软件独立成为经济发展的一支力量，还只是近二三十年的事。因此，相关的法律、商业惯例和社会规范等机制还来不及完善。软件是有价值的，但又是无形的。开发和维护相当规模的代码需要投入持续的艰苦劳动。与此同时，又可以没有限制地复制软件或者在全世界范围内分发它，却不发生任何成本。你可以很容易地把它改来改去，而不管怎么改，它都是看不见的。凡此种种，导致软件的所有权问题特

别棘手。本节将讨论一些这方面的问题，但没有给出任何解决方案。我只希望在这里介绍足够多的技术背景，以便你能够从多个角度来看待问题。当然，这些背景都依托于美国的法律，其他国家的情形虽然也类似，但肯定会有很多差异。

5.4.1 知识产权

知识产权指的是由个人经过发明或创作等创造性劳动得到的各种无形资产。有一些保护知识产权的法律适用于软件，但适用程度和范围各不相同。这些法律涉及商业秘密、版权、专利和许可。

商业秘密是最明显适用的。产权所有人要对财产保密，只有在签订了有法律约束力的合同（如保密协议）之后才能对他人公开。签保密协议比较简单，有时候效果也不错，但果真发生了泄密事件，帮助也不大。说到商业秘密，最典型的一个例子是可口可乐的配方——当然这不是软件行业的例子。理论上讲，就算它的秘密配方变得尽人皆知，任何人都可以生产相同的产品，但却不能把这些产品命名为“可口可乐”或“可乐”，因为这些都是注册商标，是另一种形式的知识产权。

版权保护创造性的表达。在文学、艺术、音乐和影视领域，版权是深入人心的。版权保护创意产品不被他人抄袭——至少在理论上，保护作者在一段时期内通过自己的产品获得收益的权利。过去，版权的保护期相当短，而今天在美国这个时间是作者生前加死后 70 年，其他国家就算短也短不了多少。（2003 年，美国最高法院裁定作者死后 70 年是一个“有限的”概念。严格来讲，这么说没错，而实际上呢，跟“永久”没有太大区别。）

保护数字产品的版权是很困难的，不同国家的版权法都不太一样。复制有形的 CD 和 DVD 很容易，而且复制得到的电子副本在互联网上无论再复制、分发多少份都几乎不费吹灰之力。通过加密或 DRM（Digital Rights Management，数字版权管理）等手段保护版权的尝试无一例外都失败了。有加密就有破解，即便破解不了，也可以在播放的同时重新录制（所谓“模拟方式的漏洞”）。针对盗版的法律追索对个体而言是很困难的，就算是大型组织要有效地打击盗版也并非易事。这个话题我们在第 9 章还会再谈。

版权也适用于程序。如果我写了一个程序，我就拥有它，就像我写了一本小说一样。其他人未经我的许可都不能使用我这个有版权的程序。说起来很简单，但别忘了那句话：魔鬼在细节里。假如你研究了我这个程序的行为，自己又写了一个功能相同的，那么该如何把握两者的相似程度才能不致侵犯版权呢？如果你只改了改代码格式，重新命名了每个变量，那还是算侵权。但除此之外，就不好说了，要解决问题恐怕只能破财去打一场官司。而如果你研究了我的程序，完全理解了它，然后名副其实地重新实现了一次，那应该没什么问题。事实上，在技术圈子里这叫“净室开发”（clean room development），也就是说，程序员完全没有接触或者不了解自己正在仿制软件的代码。虽然他们自己写的新代码与原始程序有相同的行为，但是可以证明没有抄袭。于是，法律问题就变成了一个证明，要证明净室的确是净室，没有谁因看过原始代码而受到污染。

专利为发明提供法律保护。专利与版权不同，版权只保护表达，即代码是怎么写的，而不保护代码中包含的原创思想。硬件专利很多，有轧棉机、电话、晶体管、激光，当然还有各种各样的加工方法、设备，以及对它们的改进。

最初，软件是不能申请专利的，因为它被认为是“数学”，故而不在专利法管辖范围内。自 20 世纪 60 年代以来，这种观点逐步发展，今天已变得混乱不堪。很多观察者（包括我）都悲观地认为软件及相关领域（如商业方法）的专利机制已经破产了。算法或程序是否可以取得专利没有明确的说法，非常随意。反对这种专利的一种论调认为，算法纯粹是数学知识，法律有明文规定说不能取得专利（即依法不能取得）。作为一个还算有数学背景的程序员，我觉得这种说法似乎不妥——尽管算法要用到数学，但算法并不是数学。（就说快速排序算法吧，放到今天很可能已经申请专利了。）另一种观点认为，很多软件专利都是显而易见的，只不过是使用计算机来完成一个既定或明确的过程而已，这些软件不应该取得专利，因为它们缺乏原创性。我不是专业人士，当然更不是律师，可我倒十分赞同这种观点。

亚马逊的“一键购买”（1-click）专利或许能作为软件专利的典型代表。1999 年 9 月，美国第 5 960 411 号专利被授予 Amazon.com 的四位发明人，包括创始人和 CEO 杰夫・贝索斯（Jeff Bezos）在内。这项专利涵盖“通过因特网下订单完成购买的方法和系统”，载明的创新之处是允许注册用户单击一次鼠标即可下订单购买。自此以

后，这个专利就成为了讨论会或法庭辩论的主题。坦白地讲，大多数程序员都会认为这个想法很明显，但法律规定的却是一项发明在发明当时应该对“具有一般专业技能的人”是“不明显的”。当时还是 1997 年，电子商务才刚刚萌芽。美国专利局已经拒绝了这项专利的一些主张，另外一些主张仍在持续申请中。在此期间，该项专利已授权给其他公司，包括苹果的 iTunes 在线商店。亚马逊已取得强制令，禁止其他公司未经允许使用“一键购买”。

许可是批准使用某种产品的法律协议。在安装软件时，大家都知道有一个环节：“最终用户许可协议”（End User License Agreement）或 EULA。一个对话框中显示着一个小窗口，里面密密麻麻全都是小字儿，在进行下一步之前必须先同意这个法律文本。多数人看到这个对话框，为了快点安装都会直接点过去，而这样一来，从原则上来说，或许从实践来讲也是如此，你就受到了协议条款的限制。

如果看一看这些条款，不难发现整个协议都是单方面的。供应商不承担任何保证和责任，事实上，甚至不承诺该软件会做任何事情。看看摘自 iTunes App Store 最终用户许可协议中的这一段：

> 许可商并未作如下保证：您使用许可应用程序时不会被干扰；许可应用程序中包含的功能或由许可应用程序履行或提供的有关服务会满足您的要求；许可应用程序或有关服务的运行不会受到干扰也不会出现错误；许可应用程序或有关服务中的缺陷会得到纠正。

如果软件伤害了你，你不能要求赔偿。要使用这个软件，必须满足一些条件，而且你同意不会对它进行反向工程或者反汇编。你不能把它带到某些国家。我的律师朋友说，只要相应的条款不是特别不合理，这种许可一般都是有效的，而且是可以强制执行的，听起来好像什么也没说。

另外还有一条规定可能会让你感到惊异，特别是如果你是在实体店或在线商店买的软件：“本软件仅许可使用，并不销售。”大多数购买行为在法律上都被称为“首次销售”，即一旦你买了一样东西，你就拥有它。如果你买了一本书，那这本书就是你的，你可以把它扔掉或者再卖给别人。当然，不能复制、传播，不能侵犯作者的版权。然而，数字产品供应商几乎都是以许可的形式“销售”他们的产品，从而保留权利并限制你

对“你的”副本能做什么。

2009 年 7 月发生了一件事儿，可以说是这方面一个典型的例子。亚马逊向其 Kindle 电子书读者“出售”了大量书籍，而事实上这些书都是许可阅读，并非卖出去的。后来，亚马逊知道自己分发了一些未经许可的书，于是就远程删除了所有 Kindle 中的这些书以“取消出售”。巧合的是，被他们召回的书中竟然有一本是乔治・奥威尔的反乌托邦小说《1984》，这真是绝妙的讽刺[①]。我敢肯定，乔治・奥威尔一定会对这个关于 Kindle 的故事感兴趣。

API 也会引发一些法律问题，主要都集中在版权方面。假设我是一个可编程游戏系统的制造商，类似于 Xbox 或 PlayStation 的制造商。我希望人们能买我的游戏机，而如果有很多为它开发的好游戏，那它一定会更好卖。我不可能自己来写所有程序，所以就精心设计了一套 API（应用编程接口），以便其他程序员可以为我的游戏机写游戏。我可能也会发布一个软件开发包，类似于微软为 Xbox 发布的 XDK，以方便程序员开发。运气好的话，我能卖出一大批机器，赚上一大笔钱，然后就可以高高兴兴地退休了。

API 实际上是服务用户与服务提供者之间的一个契约。它规定了接口两端都该做些什么——不是它的实现细节，而是明确规定每个函数在程序里可以做什么。这意味着其他人也可以扮演提供者的角色，只要制造一个与我竞争的游戏机，并提供跟我一样的 API 即可。做得好的话，所有游戏都可以在两个平台上运行。而如果竞争机型在某方面有优势，比如卖价低、外观设计吸引人，那我恐怕就要破产了。这对一心想致富的人来说可不是个好消息。

我有什么合法权益？我定义了 API，所以我应该能够通过版权保护它，其他人只有在获得我许可的前提下才能使用它；同样，对于我发布的 SDK 也应该如此。这样就有足够的保障了吗？法律问题，以及其他各种类似的问题，实际上并非固定不变的。根据立场不同，我可以为任何一方辩护。在本书写作期间，很多真实的公司身陷真

① 在《1984》里，政府审查员都配备了一个名为“记忆空洞”（memory hole）的焚烧槽，用于烧掉那些令“老大哥”难堪的新闻报道。在这个事例中，由于亚马逊删除图书并没有事先通知，许多消费者非常愤怒，而且谁也想不到自己买回来的书还能被亚马逊删除。由于被删除的书里有《1984》，就有读者借此来讽刺急于销毁对自己不利证据的亚马逊。——译者注

实的纷争（雇用了真实的律师，并且投入巨额的真金白银）以图解决这个问题。我偶尔也会以顾问的身份试图引导律师倾向于其中某一方，这个经历让我觉得很有意思，而且受益匪浅。本书有些内容就诞生于我向别人解释计算机领域的某些技术细节的过程中。但争吵双方的当事人一定会对此感到懊恼和愤怒，毕竟他们才是直接利害相关人。

5.4.2　标准

标准是对某些产品如何制造或者应该具有什么用途的准确、详细的说明。软件标准的例子涉及编程语言（即语法和语义的定义）、数据格式（如何表示信息）、算法处理（完成某个计算的特定步骤），等等。

某些标准是**事实**标准，比如 Word 软件的.doc 格式。事实标准指的是没有正式的名义，但每个人都在用。“标准”这个词最好只用于正式的说明书，通常由政府或协会等中立的团体制定和维护，规定某物如何制造和运作。标准的定义是足够完整和准确的，独立的实体可以反馈意见或提供中立的实现。我们每时每刻都受益于硬件标准，尽管我们根本想象不到有多少个硬件标准。如果我买了一台新电视，我之所以可以把它的电源插头插到我家的插座上，就是因为有标准规定插头的大小和形状以及电视和插座的电压。电视可以接收信号并显示画面，是因为广播和有线电视也有标准。而使用标准的 HDMI、USB、S-Video 数据线和连接器，我还可以把其他设备连接到电视上。然而，每台电视都有它自己的遥控器，每一部手机也都有各自的充电器，则是因为它们都还未实现标准化。

计算机领域也有各种各样的标准，字符集有 ASCII 和 Unicode，编程语言有 C 和 C++，算法有加密的和压缩的，还有通过网络交换信息的各种协议。有时候甚至还有相互竞争的标准，让人觉得有点浪费。（正如计算机科学家安迪・特南鲍姆（Andy Tanenbaum）所说：“多个标准的好处在于让人有多个选择。”）过去的例子有录像带的 Betamax 和 VHS 标准，而最近的例子是高清视频盘的 HD-DVD 和 Blu-ray Disc 标准。对于这两种情况，前者以一种标准最终胜出告终，后者则很可能两种标准共存，就像美国存在两种不兼容的手机技术一样。

标准很重要。有了标准，大家各自制造的东西才能集成到一起，多个供应商才能同台

竞技，而专有系统则会把每个人限制死。（专有系统的所有者自然愿意把人们都限制在它的平台上。）标准也有缺点——如果标准本身质量不高或者已经过时，但所有人又都被迫使用它，那它就会阻碍进步。不过，与它的优点相比，这些缺点还是能够接受的。

5.4.3 开放源代码

程序员编写的代码，无论使用的是汇编语言还是（更可能的）某种高级语言，都被称为**源代码**。而编译源代码得到的适合某种处理器执行的编码，叫做**目标码**。正如已经介绍的它们之间的其他区别一样，区别源代码和目标码看起来有点迂腐，但却非常重要。源代码是程序员可以读懂的，尽管可能得费点时间和精力。因此源代码是可以仔细研究并加以改编的，它所包含的任何创新和思想也是可见的。相对而言，目标码则经过了很大程度的转换，一般不太可能再恢复为类似源代码的形式，也无法从中提取出什么结构再加以改造，甚至连理解它都是不可能的。正因为如此，大多数商业软件只以目标码的形式分发，而源代码是重要的机密，因此说比喻也好，事实也罢，反正它会被锁得严严实实的。

开放源代码则是指另一种做法，即源代码可以被任何人自由阅读、研究和改进。

早期，大多数软件由公司开发，源代码是一般人看不到的，那是属于开发者所有的商业秘密。在麻省理工学院工作的理查德·斯托曼（Richard Stallman），曾经希望能够修改和加强自己使用的某些程序，但这些程序的源代码是别人私有的，自己根本看不到。为此，斯托曼感到很懊恼。1983 年，他发起了一个叫 GNU（即“GNU’s Not Unix”，gnu.org）的项目，致力于开发一些重要软件（比如操作系统和编程语言的编译器）的自由和开放版本。他还创办了一个非营利组织，叫自由软件基金会（Free Software Foundation）。这个组织的目标是开发那些永远“自由”的软件，也就是说这些软件不是私有的、不会受到所有权的限制。为此，自由软件的实现在分发时都必须遵守一个独创的版权许可，叫做 GNU 通用公共许可（GNU General Public License）或简称 GPL。

GPL 的序言如是说：“大多数软件及其他实用作品的许可，目的都是剥夺你分享和修改作品的自由。相比而言，GNU 通用公共许可则意在保证你分享和修改程序各个版本的自由，也就是确保该程序对所有用户来说仍然是自由软件。”GPL 规定，基于该许可的软件可以被自由使用，而如果再把它分发给其他人，则必须公开源代码，并同

样遵守“所有用户都可以自由使用”的许可。GPL 是一种强有力的许可，一些违反其条款的公司已经被禁止使用其代码，或者公开了以许可约束的代码为基础的源代码。

GNU 项目由很多公司、组织和个人支持，发布了大量软件开发工具和应用程序，这些软件全部采用 GPL 许可。其他类似的开源软件也采用类似的许可方式。很多时候，开源软件都为专有商业软件设立了标杆。比如，Firefox 和 Chrome 浏览器是开源的，大多数 Web 服务器上运行的 Apache Web 服务器软件也是开源的，Android 手机操作系统也是开源的。所有主要的编程语言都有开源的编译器，大多数程序员工具（包括我用来生成本书的工具）也都有开源版本。

Linux 操作系统或许是最广为人知的开源系统了（虽然它并不属于 GNU 项目），它被个人和大型商业企业广泛使用，比如谷歌的全部基础设施都运行在 Linux 之上。如果你想得到 Linux 内核源代码，访问网站 kernel.org 即可免费下载。下载后既可以自己使用，也可以对它进行任意修改。不过，要是你想以任何形式再次发布（比如，把它作为操作系统放到一个小工具里发布），那必须遵守相同的 GPL 协议开放源代码。

开放源代码是很值得研究的。把源代码送人还怎么赚钱呢？为什么程序员愿意为开源项目做贡献呢？志愿者编写的开源软件比大型专业团队协作开发的专有软件更好吗？源代码可以随便下载会不会威胁到国家安全？

这些问题持续吸引着经济学家和社会学家，也有一些答案慢慢浮出了水面。例如，红帽子（Red Hat）是一家在纽约证券交易所公开上市的公司，该公司 2011 年的年收入近 10 亿美元，市值超过 80 亿美元。他们发布的 Linux 源代码可以在网上免费下载，但公司通过支持、集成和其他收费服务可以获得收入。一些开源程序员本身就在那些使用并支持开源软件的企业工作，IBM 是一个明显的例子，但绝非特例。这些公司通过影响开源软件的发展，通过让其他人修复 bug 和改进功能而获得收益。

并不是所有开源软件都能独领风骚，开源版本不如它所模仿的商业版本的情况也比比皆是。但是，对于一些核心的程序员工具和系统来说，开源软件生生不息，的确很难被比下去。

第 6 章

软件系统

本章我们来介绍两种主要的软件：操作系统和应用程序。**操作系统**是软件中的基础层，它负责管理计算机硬件，并为其他被称作**应用程序**的程序运行提供支持。

在你使用的电脑中，无论是家里、学校里还是办公室里的个人电脑，都会装有各种各样的程序，比如浏览器、文字处理器、音乐播放器、DVD 播放器、所得税计算器（:)）、病毒扫描程序、大量的游戏，还有帮你搜索文件或查看目录的小工具。

这些程序有一个专业的叫法，即应用程序（application）。典出何处？或许出自“这个程序是计算机在完成某个任务方面的应用”吧。虽然有点啰嗦，但在称呼那些相对独立且专注于某项任务的程序时，它却是一个标准术语。这个词过去只在计算机程序员的圈子里流行，但随着苹果公司（销售在 iPhone 和 iPad 上运行的应用程序的）App Store 的迅速走红，其简写形式“应用”（app）俨然成了时尚文化的一个基本元素。

刚买的新电脑通常都会预装大量这种程序。而随着时间推移，你还会不断购买或从网上下载新程序。对我们这些用户来说，应用非常重要，而且从不同的技术角度来看，它们还具备很多有意思的性质。本章会先介绍几个例子，然后再具体讨论一下浏览器。浏览器是一个很有代表性的例子，大家都很熟悉，但它仍然会给我们带来一些惊喜，特别是它与操作系统已经呈现出分庭抗礼的态势。

闲话少叙，言归正传。还是从为应用程序运行提供支持的程序——操作系统开始讲起吧。

6.1　操作系统

20 世纪 50 年代初，还没有应用程序与操作系统之分。计算机的能力非常有限，每次只能运行一个程序，这个程序会接管整台机器。而程序员要使用计算机，运行自己的程序，必须事先预约时间段（身份低微的学生只能预约在半夜）。随着计算机变得越来越复杂，再靠非专业人员使用它们效率就会很低。于是，操作计算机的工作就交给了专业操作员。计算机操作员的任务就是把程序输入计算机，然后把计算结果送交相应的程序员。操作系统最初就是为了代替人工操作员完成上述工作才诞生的。

硬件不断发展，控制它们的操作系统也日益完善。而随着硬件越来越强大、越来越复杂，就有必要集中更多的资源来控制它们。第一批广泛使用的操作系统诞生于 20 世纪 50 年代末、60 年代初。这些操作系统通常是由硬件厂商提供的，IBM 和 Univac 都推出过自己的操作系统。后来，就连小一点的公司像 Digital Equipment 和 Data General 也都开发过自有操作系统。

操作系统也是很多大学和业界实验室的研究目标。MIT（麻省理工学院）作为这方面的先驱，在 1961 年开发了一个名为 CTSS（Compatible Time-Sharing System，兼容分时系统）的系统，该系统比同时代与之竞争的其他产品都先进得多，用起来的感觉也非常好。1969 年，贝尔实验室的肯·汤普森（Ken Thompson）和丹尼斯·里奇（Dennis Ritchie），结合他们对 CTSS 以及更完善但却不那么成功的 Multics 系统的第一手经验，开始着手开发 Unix。今天，除了微软开发的那些操作系统之外，大多数操作系统要么源自当初贝尔实验室的 Unix 系统，要么是与 Unix 兼容但独立分发的 Linux 版本。里奇和汤普森因为开发了 Unix 而一起荣获 1983 年图灵奖。

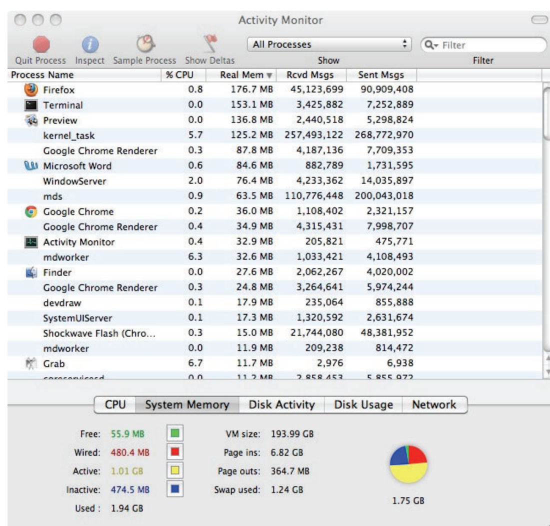
现代的计算机确实是一个复杂的“怪物”。它由很多部件组成，包括处理器、内存、磁盘、显示器、网卡，等等。为了有效地使用这些部件，需要同时运行多个程序，其中一些程序等着某些事件发生（如网页下载），另一些程序则必须实时作出响应（跟踪鼠标移动或在你玩游戏的时候刷新显示器），还有一些会干扰其他程序（启动新程序时需要在已经很拥挤的 RAM 中再腾出空地儿来）。简直就是一片混乱。

要管理如此复杂的局面，唯一的办法就是用程序来管理程序，这也是让计算机自己帮

自己的又一个例子。这么一个程序就叫作操作系统。家用和商用计算机中最常见的操作系统是微软开发的各种版本的Windows。我们日常见到的计算机90%都是由Windows管理的。苹果电脑运行的是 Mac OS X，它是一种 Unix 变体，也是在消费领域中仅次于 Windows 的第二大操作系统。而很多做幕后工作的计算机（当然也有一些直接面向用户的计算机）运行的是 Unix 或 Linux。手机中也有操作系统，它们是精简版 Windows、Unix、Linux 或其他特殊系统。例如，iPhone 和 iPad 运行的 iOS 就源自 Mac OS X，而我的 Android 手机、电视机、TiVo、亚马逊 Kindle 和巴诺 Nook 运行的都是 Linux 操作系统。我甚至可以登录自己的 Android 手机，在上面运行标准的 Unix 命令。

操作系统控制和分配计算机资源。首先，它负责管理 CPU，调度和协调当前运行的程序。它控制 CPU 在任意时刻执行的程序，包括应用程序和后台进程（如杀毒软件和检查更新的程序）。它会将一个暂时等待的程序（比如等待用户在上面单击的对话框）挂起。它会阻止个别程序多占资源。如果一个程序占用 CPU 时间太多，操作系统会强行将其中断以便其他任务得以正常执行。

操作系统通常都需要管理数十个同时运行的进程或任务。其中有些是由用户启动的程序，但大多数还是一般用户看不到的系统任务。在 Mac OS X 上通过 Activity Monitor，或在 Windows 上通过**任务管理器**，可以看到系统当前都运行有哪些程序。下图是我正在用的 Mac 中 Activity Monitor 界面的一小部分：



在上面的截图中，进程是按照它们使用的 RAM 多寡排序的。Firefox 在最上面，但几个 Chrome 进程加一块占用的内存也差不多。没有进程使用过多的 CPU。

其次，操作系统管理 RAM。它把程序加载到内存中以便执行指令。如果 RAM 空间不足，装不下所有程序，它就会将某些程序暂时挪到磁盘上，等有了空间之后再挪回来。它确保不同的程序相互分离、互不干扰，即一个程序不能访问分配给另一个程序或操作系统自身的内存。这样做既是为了保持清晰，同时也是一种安全措施，谁也不想让一个流氓程序或错误百出的程序到处乱窜。（Windows 中常见的“蓝屏死机”现象就是因为这种保护做得不到位造成的。）

为了有效利用 RAM，必须事先进行周密设计。一种思路是在必要时把程序的一部分加载到 RAM，而在程序处于非活动状态时再把它转存回磁盘，这个过程称为**交换**（swapping）。程序编写得就好像整台计算机都归它自己使用一样，而且也不必考虑 RAM 的限制。这样就大大简化了编程工作。另一方面，操作系统必须支持这种“假象”，方法就是在内存地址转换硬件的帮助下，不断地换入换出程序块，让程序认为它一直都在访问真实内存中的真实地址。这种机制被称为**虚拟内存**（virtual memory）。所谓“虚拟”，意思就是营造一种假象，而实际上这些内存并不存在。

第三，操作系统管理存储在磁盘上的信息。**文件系统**是操作系统中的一个主要组成部分，负责提供我们在计算机中都见过的那种文件夹和文件般的分层机制。本章后面还会再讨论文件系统，因为其中值得展开讨论的地方实在太多了。

最后，操作系统管理和协调外接设备的活动。它维护屏幕上的多个窗口，确保每个窗口都能显示正确的信息，而且在这些窗口被移动、缩放或隐藏后再次显示时，都能准确地恢复原貌。它把键盘和鼠标的输入送往需要这些输入的程序。它处理通过有线或无线网络连接进进出出的流量。它将数据发送给打印机和 DVD 刻录机，从扫描仪取得数据。

请注意，我说过操作系统也是程序。它跟我们在上一章讲到的其他程序一样，都是用同一类编程语言编写的，最常用的是 C 和 C++。早期的操作系统很小，因为工作比较简单。最早的操作系统每次只运行一个程序，所以程序交换量很小。没有太多内存可供分配（最多也就几百 KB）也决定了内存管理很简单。而且也没有太多外部设备需

要管理，跟今天的外设比起来显然要少得多。今天的操作系统已经非常庞大（动辄包含数百万行代码）非常复杂了，因为它们的任务本身就非常复杂。

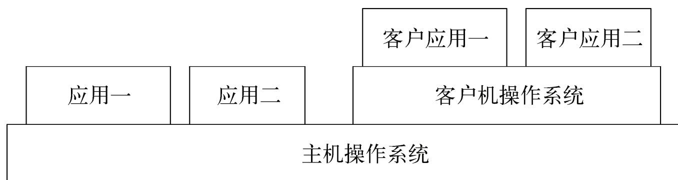
就以 Unix 操作系统第 6 版为例，它是今天很多操作系统（不包括 Windows）的鼻祖。它在 1975 年的时候是一个包含 9000 行 C 代码的汇编程序，两个人（肯·汤普森和丹尼斯·里奇）就可以把它写出来。如今的 Windows 7 拥有大约 1 亿行代码，Linux 的代码也超过了 1000 万行，它们都是几千人历经几十年工作的成果。当然，就这么直接拿来比也不太合适，毕竟现在的计算机要复杂得多，而且今天的环境和设备也要复杂得多。操作系统包含的组件同样也有差别。

既然操作系统也是程序，那么从理论上说你也可以写出自己的操作系统来。事实上，Linux 最早就是由芬兰大学生李纳斯·托沃兹（Linus Torvalds）在 1991 年写出来的，他当时的想法就是从头编写一个自己的 Unix 版本。他在互联网上发布了自己写的一个简陋程序，邀请别人试用和帮忙。自那时起，Linux 就逐渐成为软件业中一支重要的力量，很多大大小小的公司都在使用。上一章提到过，Linux 是开源软件。今天，Linux 除了核心的全职开发人员之外，还有数千名志愿者参与开发，而托沃兹负责总体把控和最终裁决。

你可以在自己的计算机中运行不同的操作系统，不一定是买来时就安装好的那一种。比如，在原来运行 Windows 的电脑上就可以运行 Linux。你可以在磁盘上存储多个操作系统，并在启动电脑时决定运行哪一个。通过苹果的 Boot Camp 可以体验这种“选择启动”功能，它能让你的 Mac 启动时运行 Windows 而非 Mac OS X。

我们甚至可以在一个操作系统的控制下运行另一个**虚拟操作系统**。使用 VMware、Parallels 和（开源的）Xen 等虚拟操作系统软件，可以在一台 Mac OS X 主机上运行另一个客户操作系统，比如 Windows 或 Linux。主机操作系统会拦截客户操作系统的请求，代替它执行那些需要具备操作系统级权限才能执行的操作，如访问文件系统或网络。主机在执行完操作后，将结果返回给客户机。在主机和客户机系统都是为相同硬件编译的情况下，客户系统大多数时候都得到硬件的全速支持，响应的及时性给人感觉就像在裸机上运行一样。

下图是一个虚拟操作系统在主机操作系统上运行的示意图。对主机操作系统而言，客户机操作系统就是一个普通的应用程序。



虚拟操作系统引发了一些有趣的所有权问题。如果某公司在一台实际的计算机上运行大量的虚拟 Windows 实例，它需要从微软购买多少 Windows 许可证？从技术上讲，只要买一个就可以。但微软的 Windows 许可中限制了可以合法运行的虚拟实例数，超过这个数目就得额外掏钱。

这里有必要说一说“虚拟”这个词的另一种用法。一个模拟计算机的程序，无论它模拟的是真实的计算机还是想象中的计算机（比如本书前面提到的玩具计算机），经常也被称为**虚拟机**。换句话说，计算机只以软件形式存在，而这种软件的行为就如同硬件一般。这种虚拟机很常见。浏览器都有一个虚拟机用于解释 JavaScript 程序，所有 Java 程序也都是通过虚拟机来解释的，而每台 Android 手机上同样有一个类似的 Java 虚拟机。

6.2　操作系统怎么工作

CPU 的结构是经过特殊设计的。计算机加电后，CPU 会开始执行存放在非易失性存储器中的一些指令。这些指令继而从一小块闪存中读出足以运行某些设备的代码。这些代码在运行过程中再从磁盘、CD、USB 存储器或网络连接的既定位置读出更多指令。这些指令再继续读取更多指令，直到加载了足够完成有效工作的代码为止。这个准备开始的过程叫做**启动**（booting），源自拉着靴带（bootstrap）给自己穿上靴子的典故。具体细节可能不同，但基本思想是一样的，即少量指令足以找到更多指令，后者依次再找到更多的指令。

计算机启动过程中通常还要检查硬件，以便知道有哪些设备接入了计算机，比如有无打印机或者无线设备。还会检查内存和其他组件，以确保它们都可以正常工作。启动过程还会为接入的设备加载软件（驱动程序），以便操作系统能够使用这些设备。上述过程都需要时间，而我们从开机到计算机能用的这段时间内通常都会等得不耐烦。尽管计算机比过去快了不知多少倍，但在启动上仍然要花一两分钟时间，的确够让人泄气的。

操作系统运行起来之后，它就会转而执行一个简单循环，依次把控制权交给准备运行或需要关注的每个应用程序。如果我在字处理程序中输入眼下这些字的时候，顺便收了一下邮件，又到网上逛了逛，同时还在后台播放音乐，那么操作系统会让 CPU 依次处理这些进程，并根据需要在它们之间切换。每个程序会得到一段极短的时间，在程序请求系统服务后或者分配给它的时间用完时结束。

操作系统会响应各种事件，比如音乐结束、邮件或网页到达，或者用户按下了键盘上的按键。对这些事件，操作系统都会作出必要的处理，通常是把相应的事件转发给相关的应用程序。如果我重新排列屏幕上的窗口，操作系统会告诉显示器把窗口放在什么地方，并告诉每个应用程序它们各自窗口的哪一部分可见，以便重新绘制窗口。如果我选择“文件 > 退出”或单击窗口右上角的“×”按钮退出应用程序，系统会通知应用程序它马上要“死”了，以便它赶紧“安排后事”（比如，弹出对话框询问用户“您想保存这个文件吗？”）。然后，操作系统会回收该程序占用的所有资源，并告诉那些窗口得见天日的其他程序，必须重绘各自的窗口了。

6.2.1 系统调用

操作系统提供了硬件和其他软件之间的接口。有了这个接口，硬件就好像能听懂人的话了，而程序员编程因此就会变得简单。用这个圈子里的行话说，操作系统提供了一个**平台**，在这平台上可以构建应用程序。

操作系统为应用程序定义了一组操作（也叫服务），比如将数据存储至文件或者从文件中取出数据、建立网络连接、获取键盘输入、报告鼠标移动和按钮点击、绘制屏幕，等等。

操作系统以标准化的或者说大家协商一致的方式提供这些服务，而应用程序通过执行一种特殊的指令来请求这些服务，并将控制权移交给操作系统中特定的地址。操作系统根据请求完成计算，然后再将控制权和结果返回给应用程序。操作系统的这些“入口”被称为**系统调用**（system call），而对这些系统调用的详细说明实际上恰恰解释了操作系统能做什么。系统调用可以直接拿操作系统内部的代码作为入口，也可以是对某个（为相应服务而准备的）库函数的调用。但多数情况下，即便是程序员也不用关心上述区别。正因为如此，谁也说不清楚到底有多少个系统调用，但通常一两百个总是有的。

6.2.2 设备驱动程序

这一节我们讲一讲操作系统中的另一个部件——设备驱动程序。**设备驱动程序**是一种沟通操作系统与特定硬件设备（如打印机和鼠标）的程序。驱动程序的代码知道怎么让特殊的设备履行自己的职责，比如从特定的鼠标得到移动和按钮信息、让磁盘通过旋转的磁表面读取和写入信息、让打印机在纸上留下记号、让特定的无线网卡发送和接收无线电信号。

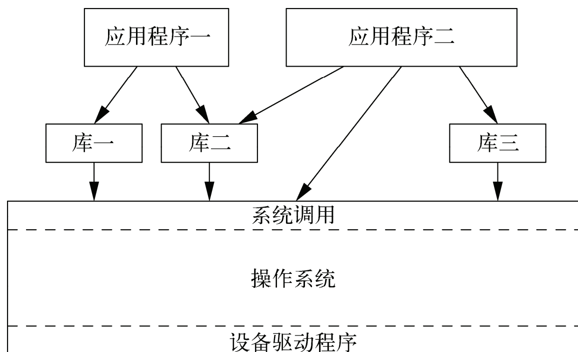
就说打印机吧。操作系统只会发出标准的请求，比如“在这个位置上打印这段文本”、“绘制这幅图像”、“移到下一页”、“描述你的能力”、“报告你的状态”，等等。而且，还是以适合所有打印机的标准方式发出这些请求。然而，打印机的能力是有差别的，比如支不支持彩色打印、双面打印，或者不同纸张大小。打印机专属的驱动程序，要负责把操作系统请求转换为特定设备完成相应任务必需的指令。一句话，就是操作系统发送通用的请求，而具体的设备驱动程序负责在各自硬件上落实、执行请求。

驱动程序把操作系统与特定设备独有的性质隔离开来（任何设备，比如各种键盘，都有一些操作系统要用到的基本性质和操作），操作系统通过驱动程序的接口以统一的方式访问相应设备，从而方便在设备之间切换。

通用的操作系统都包含很多驱动程序。例如，Windows 为满足各种潜在用户的需要，在发行时就已经带有各种各样的设备驱动程序。每个设备的制造商都有自己的网站，提供新版本或更新的驱动程序下载。

启动过程中有一个环节就是把当前可用设备的驱动程序加载到运行的系统中。可用的设备越多，加载要花的时间就越长。新设备随时有可能出现。在把外部磁盘插入 USB 插槽后，Windows 会检测到这个新设备，（根据设备驱动程序接口的某个部分）确认它是一个磁盘，然后加载 USB 磁盘驱动程序与这个磁盘通信。Mac OS X 操作系统也一样。一般来说，没有必要不断升级新的驱动程序，因为所有设备的接口都是标准化的，操作系统本身已经包含了必要的代码，而驱动设备的特殊程序也已经包含在设备自身的处理器中。

下面这幅图展示了操作系统、系统调用、驱动程序和应用程序之间的关系。



6.3 其他操作系统

随着各种电子元器件越做越小，价格越来越便宜，一台设备所能集成的电子元器件数量也越来越多。于是，很多设备都具有了很强大的处理能力和很充足的内存空间。正因如此，要说一部数码相机是“一台带镜头的计算机”也不为过。随着处理能力和内存容量的增加，相机的功能也越来越强大。手机不也一样嘛！当然啦，手机和相机现在都已经合二为一了。今天，随便一部手机上的相机所拥有的像素解析度，都远远超出了早期的数码相机。没错，镜头的品质得单说。

总之，这些设备俨然与主流通用计算机一般无二。它们都有强劲的处理器、大容量的内存，以及一些外围设备（比如数码相机上的镜头和显示屏）。它们的用户界面极其精美。它们可以通过网络连接与其他系统通信（手机使用电话网络和 Wi-Fi，游戏机手柄使用红外线和蓝牙），不少还提供 USB 接口，以支持移动硬盘的临时接入。

随着这种趋势的不断演进，选择市面上现成的操作系统要比自己从头写一个来得更实际。除非用途特殊，否则在 Linux 基础上改一改是成本最低的，关键是 Linux 非常稳定、容易修改、方便移植，而且免费。相对而言，自己开发一个专有系统，或者取得某个商业系统的许可，都会引入巨大开销。当然，改造 Linux 的缺点在于必须把改造后的操作系统部分代码按照 GPL 许可发布,由此可能引发如何保护设备中知识产权的问题。不过，从 Kindle 和 TiVo 的案例来看，似乎也没有什么不能解决的。

6.4　文件系统

文件系统是操作系统的一个组成部分，它能够让硬盘、CD 和 DVD、移动存储设备，以及其他各种存储器等物理存储媒体，变成看起来像是由文件和文件夹组成的层次结构。我们常说计算机有逻辑组织和物理实现两大概念，文件系统就是这两大概念的集中体现。文件系统能够在各种不同的设备上组织和存储信息，但操作系统则为所有这些设备都提供相同的接口。文件系统存储信息的方式以及存储多久，最终甚至会衍生出一些法律问题来。所以说，研究文件系统的另一个目的，就是要理解为什么“删除文件”并不代表其内容会永远消失。

几乎所有人都用过 Windows 的资源管理器或者 Mac OS X 的 Finder，这两个工具都能列出自最顶层（比如 Windows 中的 C 盘）开始的文件系统的层次结构。在这个层次结构里，一个**文件夹**（folder）可以包含其他文件夹和文件。换句话说，点开一个文件夹，就可以看到更多文件夹和文件。（Unix 系统则一直使用**目录**（directory）而不是文件夹的概念。）这里的文件夹是一个组织结构的概念，而实际的文档内容、图片、音乐、电子表格、网页等，则保存在文件中。在计算机中，所有这些信息都存储在文件系统内，只要用鼠标点击几下就可以找到。文件系统中不光存储数据，还存储着可以执行的程序（比如浏览器）、代码库、设备驱动程序，以及构成操作系统自身的文件。这些文件的数量大得惊人，就说我这个普普通通的 MacBook 吧，其中存储的文件已经超过了九十万个。

文件系统管理所有这些信息，方便其他程序和操作系统的其他部件读写这些信息。它统筹安排所有的读写操作，确保这些操作有效进行，且不会相互干扰。它记录数据的物理位置，确保它们各就各位，不会让你的电子邮件意外地窜到你的电子表格或纳税申报单里面去。在支持多用户的系统中，还要保证信息的隐私权和安全性，不能让一个用户在未经允许的情况下访问另一个用户的文件。另外，可能还需要限制每个用户有权使用的硬盘空间，也就是所谓的配额管理。

在最低级的层次上，文件系统服务是通过系统调用来提供的。程序员通常要借助代码库来使用这些系统调用，以简化编程过程中常见的文件处理操作。

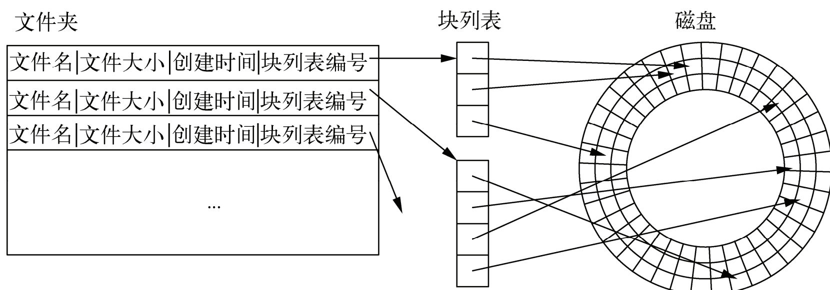
6.4.1 磁盘文件系统

无论什么样的存储设备，在文件系统中一律只表现为文件夹和文件。从这一点来说，文件系统是把物理实现抽象为逻辑组织的绝佳范例。但它又是怎么做到的呢？

一块 100 GB 的硬盘可以存储 1000 亿字节的数据，但这块硬盘上的软件可能会将其看成 1 亿个 1000 字节大的**块**。（现实中，块的大小应该是 2 的幂，而且每个块还会更大。这里使用十进制是为了便于说明其中的关系。）这样，一个 2500 字节大的文件（比如一封普通的邮件），就需要 3 个这样的块来存储。因为 2 个块存不下，而 3 个块有富余。文件系统不会在同一个块内存储不同文件的信息，因而就免不了有一些浪费。因为存储每个文件的最后一个块不会完全用完（在我们举的这个例子中，最后一个块就会闲置 500 字节）。考虑到简化记录工作所节省的工作量，这点代价还是值得的，更何况磁盘存储器已经那么便宜了。

这个文件所在的文件夹条目中会包含文件的名字、2500 字节的文件大小、创建或修改时间，以及其他细节信息（权限、类型等，取决于操作系统）。所有这些信息都可以通过资源管理器或者 Finder 看到。

这个文件夹条目中还会包含文件在磁盘上的位置信息，也就是 1 亿个块中的哪 3 个块存储着这个文件。管理这些位置信息的方法有很多种。比如，文件夹条目可以包含一组块编号，也可以引用一个自身包含一组块编号的块，或者只包含第一个块的编号，第一个块又包含第二个块的编号，依此类推。下面这幅示意图展示了文件夹引用块列表的大致情况：



存储同一个文件的块在磁盘上不一定连续。事实上，这些块一般都不挨着，至少存储

大文件的块是这样的。兆字节级别的文件需要占用上千个块，这些块通常会分散在磁盘的各个地方。尽管从这幅图中看不出来，但文件夹及块列表本身也存储在块中。

文件夹也是一个文件，只不过这个文件中包含着文件夹和文件的位置信息。由于涉及文件内容和组织的信息必须精准、一致，所以文件系统保留了自己管理和维护文件夹内容的权限。用户和软件只能请求文件系统来间接地修改文件夹内容。

没错，文件夹也是文件。从存储方式上讲，它们跟文件没有任何区别。只不过文件系统会全权负责管理文件夹内容，任何应用软件都不能直接修改该内容。除此之外，它们都保存在硬盘上的块中，由相同的机制进行管理。

在应用程序要访问已有的某个文件时，文件系统必须从其顶级层次开始搜索该文件，在相应文件夹里查找文件路径中的每一部分。举个例子，假设要在 Mac 中查找 /Users/bwk/book/book.txt。文件系统首先要在其顶层搜索 Users，然后在该文件夹里搜索 bwk，接着在找到的文件夹里搜索 book，最后再在找到的文件夹里搜索 book.txt。在 Windows 中，这个文件的路径可能是 C:\My Documents\book\book.txt，但搜索过程相似。这是一种化整为零的思路。也就是说，路径中的层次会逐步缩小要搜索的文件或文件夹的范围，同时把其他不相干的部分过滤掉。正因为如此，不同层次中的文件可以使用相同的名字，唯一的要求是完整的路径必须独一无二。实践中，应用程序和操作系统会记住当前的文件夹，因而文件系统不必每次都从顶层开始搜索。而为了加快处理速度，系统还可能会缓存频繁用到的文件夹。

应用程序在创建新文件时会向文件系统发送请求，文件系统会在相应的文件夹中增加一个新条目，包含文件名、日期等项，还有文件大小为零（因为还没有为这个新文件分配磁盘块）。接下来，应用程序要向文件中写入某些数据时（比如向一封邮件中写几句话），文件系统会找到足够多的当前没有使用的或者“空闲”的块来保存相应内容，并把数据复制过去。然后把这些块插入到文件夹的块列表中，最后返回给应用程序。

不难想象，文件系统还要维护一个磁盘上当前未被使用（也就是还没有成为某些文件一部分）的块的列表。每当应用程序请求新磁盘块，它就可以从这些空闲的块中拿出一些来满足请求。这个空闲块的列表同样也保存在文件系统的块中，但只能由操作系统访问，应用是访问不到的。

6.4.2 删除文件

删除文件时，过程恰好相反：文件占用的块会回到空闲列表，而文件夹中该文件的条目会被清除，结果就好像文件被删除了一样。

现实中的情况并不完全如此，而是加入了一些有意思的比喻。当你在 Windows 和 Mac OS X 中删除一个文件时，这个文件会跑到“回收站”或“垃圾桶”里去。“回收站”和“垃圾桶”不过是另外一个文件夹，但具备某些常规文件夹所不具备的属性。正因为如此，才成其为“回收站”嘛。删除文件时，相应的文件夹条目将从当前文件夹被复制到名叫“回收站”或“垃圾桶”的文件夹里，然后会清除掉原来的文件夹条目。但是，这个文件占用的块以及其中的内容没有丝毫变化！从“回收站”里还原文件的过程正好相反，就是把相应条目恢复到它原来所在的文件夹中。

“清空回收站”倒是跟我们本节一开始描述的过程很相似。此时“回收站”或“垃圾桶”里的文件夹条目会被清除，相应的块会真正再添加到空闲块列表中。不管是明确地执行这个操作，还是文件系统因为空闲空间过少而在后台静默地清空，这个过程都将实实在在地发生。

假设是你明确地执行清空操作。那么这个操作首先清除“回收站”文件夹中的条目，然后把其中文件占用的块回写到空闲块列表。但是，这些文件的内容并没有被删除。换句话说，原始文件占用的每个块中的所有字节都会原封不动地呆在原地。除非相应的块从空闲块列表中被“除名”并奉送给某个应用程序，否则这些字节不会被新内容覆盖。

这意味着什么呢？意味着你认为已经删除的信息实际上还保存在硬盘上。如果有人知道怎么读取它们，仍然可以把它们读出来。任何可以不通过文件系统而能够逐块读取硬盘的程序，都可以看到那些被“删除”的内容。

显然，这样有一个潜在的好处。就是在硬盘出问题的情况下，还有可能恢复其中的信息，尽管文件系统可能已经一团糟了。可是不能保证数据真正被删除也有问题。假如你想删除的文件里包含隐私，甚至一些见不得人的东西，你肯定希望它们被删除后永远销声匿迹。对精于此道的坏蛋或者执法机关的专家来说，恢复磁盘中的内容只是小菜一碟。因此，假如你在文件中记录了自己穷凶极恶的想法，或者在妄想症支配下写

了很多胡话，那最好使用能够把这些信息从空闲块中彻底擦干净的程序。比如 Mac 中的“安全擦除”选项在释放磁盘块之前，会先用随机生成的比特重写其中的内容。

现实当中的你还应该知道更加保险的做法。因为即使用新信息重写了原有内容，一名训练有素的敌人仍旧可以凭借他掌握的大量资源发现蛛丝马迹。军事级的文件擦除会用随机的 1 和 0 对要释放的块进行多遍重写。更为保险的做法是把整块硬盘放到强磁场里进行消磁。而最保险的做法则是物理上销毁硬盘，这也是保证其中内容彻底销声匿迹的唯一可靠方法。如果你的磁盘一直都在执行自动备份（就像我在上班时使用的计算机一样），或者你的文件保存在网络文件系统中而不是本地硬盘上，那么这些招数恐怕也都不灵光了。

文件夹条目本身也存在类似的问题。删除一个文件时，文件系统会让相应文件夹条目不再指向有效的文件。为此，它可能只会把一个表示“本条目不再使用”的比特位设置为 1。这样在将来需要恢复该文件的原始信息，包括所有未被重新分配的块的内容时，只要把这个比特位重置为 0 就可以了。事实上，1980 年代微软 MS-DOS 中的文件恢复系统采用的就是这种办法。对于待释放的空闲条目，该系统会把相应文件名的第一个字符设置为一个特殊值。这样，如果用户很快又要恢复文件，那么实现起来会简单很多。

知道了文件在被创建人删除后还可能存在很长时间，有助于我们理解一些法律程序，比如当事人坦白和文档保全的意义。在法庭上，这种案例屡见不鲜。有时候，一封陈年邮件就可能影响对被告的量刑，至少会让犯罪嫌疑人的陈述露出马脚。如果这些记录只存在于纸面上，那么徒手将其撕碎就能轻易销毁证据。但数字化记录是会扩散的，还可能通过移动硬盘等媒体藏匿于很多地方。（维基解密 2010 年得到的大批机密外交文件就保存在很多张 CD 中。）明智的人都应该时刻注意自己在邮件里的措辞，甚至应该注意通过计算机发表的任何言论。

6.4.3 其他文件系统

刚才我们讨论的是硬盘驱动器（包括移动硬盘）上的常规文件系统。我们大多数的信息都保存在这些硬盘上，而且我们对它们也非常熟悉。不过，这个文件系统也同样适用于其他媒体。

例如，已经退出历史舞台的软盘，在逻辑上具有同样的层次结构，但细节上有所不同。CD-ROM 和 DVD 同样以文件系统的方式提供访问界面，同样由不同层次的文件夹和文件组成，只不过一般为只读，不能写。固态硬盘通过闪存来模拟常规硬盘，但重量更轻，耗电更省。

USB 闪存盘和 SD（Secure Digital，安全数字式）闪存卡已经无处不在。把它们插入到一台 Windows 计算机中，它们就会像一块新硬盘一样。通过资源管理器可以查看其中的内容，并像在普通硬盘上一样执行读写操作。唯一的区别就是它们容量小一些，有时候速度可能也慢一些。

如果把它们插到一台 Mac 上，它们同样表现为分层的文件系统，可以通过 Finder 浏览，文件也可以拷来拷去。把它们插到 Unix 或 Linux 计算机上也一样，它们还是表现为文件系统。让这些硬件在不同操作系统中看起来具有同样的文件系统和同样的文件夹/文件结构的是软件。但在内部，文件组织采用的可能是微软的 FAT 系统，其他文件系统也都模仿该系统。但我们不需要去理会这个，这种抽象是非常完美的。（顺便说一句，FAT 是 File Allocation Table 的简写，即“文件分配表”，不是“肥胖”的意思。所以大家可别误以为微软的实现不好。）



我的第一台数码相机都把照片存在自己内部的文件系统中。为了拷照片，必须用数据线把相机连接到计算机。而我现在这台相机使用的是 SD 闪存卡（就是上面图里的那张），把这张卡从相机里拔出来插到计算机上，就可以上传照片。不仅上传速度比以

前快得多，而且也让我摆脱了以前相机厂商开发的那些难用至极的滥软件。熟悉而统一的界面代替了笨拙而专有的方法。显而易见，相机厂商也会因此省不少事，至少不用再为自己的产品开发专门的文件传输软件了。

值得一提的是，同样的思想也体现在网络文件系统上。在学校和公司里，把文件保存到服务器上是非常常见的做法。借助相应的软件，我们访问其他计算机上的文件系统时，就如同访问本地的硬盘一样。同样只要使用资源管理器、Finder 或者其他软件就可以。远端的文件系统可能与本地相同（比如两台 Windows 计算机），也可能不同（比如其中一台是 Mac 或 Linux 计算机）。但与闪存设备一样，软件把它们的差异隐藏了起来，我们看到的永远是与自己本地计算机中常规文件系统一样的界面。

网络文件系统经常用于备份，当然也可以作为主文件存储系统。必要时，可以把旧文件复制到便于存档的媒体上，保存到其他地方，以免发生火灾等事故时毁坏重要资料。有些磁盘系统会依赖一种叫 RAID（Redundant Array Of Independent Disks，独立磁盘冗余阵列）的技术，把数据和错误校验码分别写到多个磁盘上，以便某个磁盘损坏时能够从其他磁盘恢复数据。当然，这种系统也会增加彻底销毁数据的难度。

6.5 应用程序

“应用程序”是一种统称，表示所有在操作系统平台上完成某种任务的软件或程序。应用程序可大可小，可以只完成特定的任务，也可以囊括大量功能。可以是花钱买的，也可以是免费送的。它的代码可以高度保密，也可以开放源码，甚至没有任何限制。

或许可以把应用程序分成两类。一类是小型独立的应用，通常只帮用户做一件事；另一类是大型软件，包含非常多的操作，比如 Word、iTunes 或 Photoshop。

最简单的应用程序是啥样的？好吧，我们以 Unix 中的 `date` 程序为例，它的作用就是打印当前日期和时间：

```
$ date
Thu Sep 22 19:44:07 EDT 2011
```

date 程序在所有类 Unix 系统中的行为完全一样。不信？你可以在 Mac 里打开 Terminal（终端）窗口试一试。date 的代码很少，因为它是构建在相应系统调用和库函数基础上的。系统调用为它提供了以内部格式表示的日期和时间，库函数帮它转换格式并打印出来。以下就是实现它的 C 代码，你看短不短：

```
#include <stdio.h>
#include <time.h>
int main() {
    time_t t = time(0);
    printf("%s", ctime(&t));
    return 0;
}
```

Unix 系统有一个列出目录中文件和文件夹的命令，它是 Windows 资源管理器和 Mac OS X Finder 的纯文本版。对文件执行复制、移动等操作的程序，在 Finder 和资源管理器中也都有对应的图形用户界面版。同样，这些程序也使用系统调用来提供文件夹包含内容的基本信息，也依赖于库函数去读、写、格式化和显示信息。

Word 之类的应用程序比浏览文件系统的程序要大得多。但很明显，Word 一定包含某种类似的文件系统程序，以便用户能够打开文件、读取文件内容和保存文档。Word 也包含非常完善的算法，随着文本变化持续更新显示界面的算法就是一例。它还提供精心设计的用户界面，用于显示信息和让用户调整字号、字体、颜色、布局等的各种选项。对这种程序而言，用户界面是至关重要的一部分。Word 以及其他具有巨大商业价值的大型程序都经历了不断改进和功能完善。我还真不知道 Word 有多少行代码，但要说它有几百万行 C 和 C++代码应该一点都不奇怪。

另一个大型、免费，有时候甚至是开源的应用程序是浏览器。从某种角度说，浏览器的复杂度甚至更高。只要上过网，你就至少使用过下列浏览器之一：Firefox、Safari、Internet Explorer、Chrome、Opera。相信不少读者也像我一样，用过其中不止一个。第 10 章我们会更详细地介绍 Web 和浏览器如何获取信息。这里我们只关注它是一个多大、多复杂的程序。

从外部来看，浏览器会向 Web 服务器发送请求，从那里取得信息后再把它们显示出来。那它复杂在哪里呢？

首先，浏览器必须处理**异步**事件。所谓异步事件，就是在非预定时间发生、没有特定次序的事件。举个例子，你点击一个链接，浏览器就会发送一个对相应页面的请求。

但发送完请求后，它不能就那么一直等着。它还得准备响应你的其他操作，比如滚动当前页面，或者在你点击“后退”按钮或另外一个链接时中断之前的请求，不管请求的页面是否已经到达。在你调整窗口大小时，它必须不断更新窗口中的内容，或许就因为你在等待新页面期间没事儿干，于是就会随手来回缩放起窗口来。如果页面中包含音频和视频，那它还要负责控制它们。编写异步系统一直是非常困难的，而浏览器就涉及很多异步操作。

浏览器必须支持很多种内容，包括静态文本和具有互动性的程序（可以动态改变网页中包含的内容）。对某些内容的支持可以委托给辅助程序（这是处理 PDF 文档和电影的标准做法），但浏览器本身必须提供相应的机制，以便启动这些辅助程序，为它们发送和接收数据以及请求，还要控制它们。

浏览器必须管理多个标签页或窗口，每个标签页和窗口都可能需要执行前述操作。它要为每个标签页和窗口单独保留一份历史记录，还要保存书签、收藏夹等数据库。它要支持访问本地文件系统，以便上传和下载文件。

浏览器自身还是一个平台，要提供不同层次的扩展接口。比如，要支持 Flash 和 Silverlight 插件、JavaScript 和 Java 虚拟机，以及 Firefox、Safari 和 Chrome 所支持的那些扩展程序。

浏览器既然包含那么多实现复杂功能的代码，其自身以及它所支持的插件、扩展程序免不了会存在一些漏洞，面临被攻击的风险。另外，一些无知、愚昧，甚至白痴用户（本书读者自然不在此列），由于不理解浏览器的原理，不知道可能存在的风险，也会导致浏览器遭受攻击。总之，做浏览器开发确实不容易。

如果现在再回头读一读本节内容，你会不会想到些什么？没错，现在的浏览器非常像操作系统。它要管理资源、控制同时发生的活动，它向多个地方请求和保存资源，并且为其他程序运行提供了一个平台。

多年来的实践表明，把浏览器当成操作系统是可行的。换句话说，浏览器本身就是一个独立的系统，与什么操作系统在控制底层硬件无关。大概十几年前，这种想法已经浮出水面，但当时还存在很多实际的困难。今天，这种可能性已经触手可及。大量服务都可以只通过浏览器界面来访问了（邮件是最明显的例子），而这个趋势还在继续。

谷歌已经发布了一个浏览器操作系统，叫 Chrome OS。这个操作系统完全依赖于 Web 服务。为此，谷歌还推出了运行 Chrome OS 的计算机 Chromebook。第 11 章讨论云计算的时候，我们还会再探讨这个话题。

6.6 软件分层

与计算领域的很多其他东西一样，软件也是分层组织的。类似于地质学中的分层，软件中的不同层次可以隔离不同的关注点。在程序员的世界里，分层是解决复杂问题的一个核心思想。

通俗地讲，计算机的最底层是硬件。硬件，除了总线支持在系统运行期间添加和删除设备之外，其他方面几乎可以看成不可变的。

再往上就是所谓的操作系统层了。为了突出其核心地位，通常把这一层称为**内核**（kernel）。操作系统介于硬件和应用程序之间。无论底层是什么硬件，操作系统都要负责隐藏其特殊性，向应用程序提供统一的接口或界面，这个接口或界面不因硬件的种种差别而变化。在接口设计得当的情况下，同一个操作系统的接口完全可以适用于众多制造商生产的不同类型的 CPU。

Unix 操作系统的接口就是这样的。Unix 可以在各种处理器之上运行，但在任何处理器上都能提供相同的核心服务。事实上，操作系统就是一种通用的商品，底层的硬件除了价格和性能之外，其他方面都影响不大。而且，上层的软件也不依赖于它。把为一种处理器编写的程序移植到另一种处理器上，无非就是小心谨慎地用合适的编译器再编译一遍而已。当然，程序与硬件结合得越紧密，这种转换工作就越难做。无论如何，这种转换对很多程序来说都是可行的。举个大规模转换的例子，苹果公司在 2005 年到 2006 年，用了不到一年时间，就把它们的软件从 IBM 的 PowerPC 处理器转换到了 Intel 处理器上。

对 Windows 来说，问题就没有那么简单了。从 1978 年的 Intel 8086 CPU 开始，Windows 的开发就与 Intel 架构紧密结合，包括后来 Intel 发布的每一款 CPU。（处理器系列一般称为“x86”，是因为 Intel 的处理器很多年都以“86”这个编号结尾，包

括 80286、80386、80486 等。）Windows 与 Intel 的结合是如此紧密，以至于这样的系统一度被世人称作“Wintel”。

操作系统再往上的一层是函数库。函数库提供通用的服务，这样一来，程序员就不必各自重复实现这些功能。有些库比较靠近底层，能够完成一些基本功能（完成数学计算，比如开方和求对数，或者像前面 `date` 命令一样计算日期和时间）。另外一些库的功能更强大（涉及加密、图形处理、压缩等）。图形用户界面上的组件，包括菜单、按钮、复选框、滚动条、选项卡面板等等，都需要编写很多代码。为此，只要把这些代码封装成函数库，任何人就都可以使用它们，而且还能保证统一的行为和外观。这就是为什么大多数 Windows 应用（至少它们的基本图形组件）看起来那么相似的原因。同样的情况在 Mac 上更是如此。如果所有软件开发商都重新发明、重新实现这些功能，那不仅会浪费大量资源，而且五花八门的界面也会让用户感到无所适从。

如前所述，典型的应用程序会使用函数库和操作系统服务，把它们集成到一起实现某种功能。不过，库函数与系统调用之间的区别并不十分明显。某个特定的服务可以作为系统调用实现，也可以借助使用了系统调用的库函数来实现。

有时候，内核、函数库和应用程序之间并不像我说的那么泾渭分明。毕竟，编写及连接软件组件的方式多种多样。例如，内核可以提供少量服务，而依赖上层的库来完成大部分工作。或者，它也可以自己承担大部分任务，而较少地依赖于库。操作系统与应用程序之间并没有清晰的界限。

那该如何区分它们呢？一个简单（但可能不够完美）的方法，就是把任何确保 A 应用程序不会干扰 B 应用程序的代码看成是操作系统的职能。比如，内存管理、文件系统、设备管理和 CPU 管理，这些都是操作系统的职能。内存管理需要决定在程序运行的时候把它们放到内存的什么位置，文件系统需要决定把信息保存到磁盘上的什么位置，设备管理需要确保两个应用程序不会同时占用打印机，也不能在没有协商的情况下就向显示器输出内容。最核心的还是 CPU 管理，因为这是操作系统履行前述各项职能的前提条件。

浏览器不属于操作系统，因为你可以运行任意浏览器，甚至同时运行多个浏览器，都不会干扰共享资源或者控制流程。

听起来好像是可以做到泾渭分明似的，但要是较起真来，比如打起官司来，那可就难说了。美国司法部从 1994 年开始（到 2011 年结束的）对微软的反垄断诉讼，就涉及微软的 Internet Explorer 浏览器到底是操作系统的一部分，还是一个独立应用程序的问题。如果浏览器是操作系统的一部分（按照微软的主张），那么要求微软删除 IE 就是不合理的，而微软要求用户使用 IE 的做法就是正当的。如果浏览器是一个独立的应用程序，那么微软就涉嫌以非法手段强迫用户在非必要情况下使用 IE。当然，这个官司本身要复杂得多，但如何界定浏览器的归属问题确实非常重要。最终诉讼的结果是，法院认定浏览器是一个独立的应用程序，不属于操作系统。用法官托马斯·杰克逊（Thomas Jackson）的话说："Web 浏览器和操作系统是相互独立的产品。"

第 7 章

学习编程

在我的课上，我会教给学生们一些编程的知识，使用一种叫 JavaScript 的编程语言。生活在信息时代的人，有必要了解一些编程方面的常识。比如，哪怕是非常简单的程序，让它顺顺当当地跑起来都可能要克服难以想象的困难。通过这门课让大家知道如何驾驭计算机，乃至初步品尝到第一次运行程序就顺利通过的美妙感觉，在我看来是一件非常有成就感的事。当然，更理想的情况是在大家将来拥有了足够多的编程经验后，再听别人说编程很容易啦，或者某个程序没有任何错误等等，你会情不自禁地警觉起来。如果你自己折腾一天连 10 行代码都调试不好，那别人要是说能按时交付百万行级的程序，而且没有任何 bug，你相信吗？换个角度说，有点编程常识也能让人明白，其实也不是写什么程序都那么难，大不了请人帮你写呗。

那我们用什么语言来学编程呢？语言实在太多了，可以说不计其数。但是，没有哪门语言是最适合新手学习，或者说我觉得最适合在这样一本通俗读物里教给所有“外行”的。很早以前，我用过微软的 Visual Basic。它既是一门语言，也是一个编程环境。通过它可以方便地写出看起来很专业的 Windows 应用程序。而且，微软的 Office 办公套件以及很多其他办公软件里也都内置了某种 VB 的简化版。因此，会用 VB 就可以在日常工作中增强或更好地控制 Word 和 Excel（当然，弄不好也会为病毒入侵打开便利之门）。

可惜 Visual Basic 已经不再是最好的选择了。尽管微软提供的免费版几乎没有删减任何功能，但比起十几年前，这门语言及其体系实在复杂得太多了。更重要的是，VB

只能在 Windows 系统上运行。而我需要找一个随便在什么平台上都能跑的语言。

为此，我选择了 JavaScript，因为它具有如下优点。首先，它无处不在，所有浏览器都支持它。几乎每一个网页多多少少都有 JavaScript 程序，而且其代码也很容易向别人展示。要是你用它写了一个程序，把它放在自己的网页里就能博得朋友和家人的赞美。其次，这门语言本身比较简单，对学习者的要求很低。当然，这并不意味着它能力弱。事实上，JavaScript 非常强大，可以完成极为复杂的计算任务。很多网页特效的背后都是 JavaScript，包括谷歌的在线办公程序 Google Docs 和其他类似的应用。最后，Twitter、Facebook、Amazon 等等这些世界级的大网站都提供了 JavaScript 的 API。

JavaScript 当然也有缺点。不同浏览器中的 JavaScript 实现并不像我们想象的那样完全一致。换句话说，在一个浏览器中能运行的程序，换一个浏览器可能就运行得不正常。但就本章的学习目标而言，这根本不是问题。做专业开发的前端工程师都有办法解决这个问题。另外，这门语言的某些特性不太好理解，有时候会显得比较怪异。JavaScript 程序通常只能在网页中运行，很少能独立存在。不过也有一些软件支持 JavaScript 程序，比如 Adobe 的 PDF 阅读器。由于它委身于浏览器，因此要学它，一般都得同时学一点 HTML（HTML 是一种描述网页结构的标记语言）。尽管存在这些缺点，JavaScript 仍然非常值得学习。

只要能够理解本章的内容，你就可以学会编程，至少能掌握一些最基本的编程方法。但不管怎样，编程都是极有必要掌握的一种技能。学会了基本的编程方法之后，再学习其他语言也就不会那么难了。如果你希望学得更深入一些，或者找一些其他学习资料，可以上网搜索“JavaScript 教程”。按下回车键，你会看到一大堆结果。

不过，就算你现在不想学编程，不去理解语法甚至干脆跳过这一章也无所谓。不看本章也不会影响对其他章节的阅读和理解。

7.1　编程语言的基本概念

编程语言的某些基本概念是相通的，因为这些概念都是为了表达一系列计算步骤而发

明的。任何编程语言都会提供一些手段，用于取得赖以完成计算的输入数据、进行算术计算、在计算期间存储和获取中间值并显示结果、根据之前的计算结果决定下一个计算步骤，以及在计算完成时保存结果。

是语言就有**语法**，而语法就是一系列规则，根据它们可以判断什么符合语法，什么不符合语法。编程语言对语法规则是锱铢必较的，哪怕有一点点地方违反语法，它都会提出抗议。语言还要有**语义**，语义规定了语言中所有元素的含义。

理论上讲，一段程序的语法是否正确，以及语法正确的情况下其含义是什么，这些都不应该有歧义。但实际上，就像用自然语言写文章一样，任何歧义都没有的理想状态有时候很难达到。语言的最基本单位通常是字和词，这些字和词本身可能就有歧义，而对它们的不同理解更是司空见惯。因此，实现这些语言规则的时候可能就会有偏差。另外，语言本身也会与时俱进。综上所述，不同浏览器对 JavaScript 的实现多多少少都会存在差异。实际上，即使是同一款浏览器的不同版本，对 JavaScript 的实现也有差异。

JavaScript 这门语言实际上包含三个方面。第一是语言本身，包括让计算机完成算术计算的语句、测试条件，以及重复计算的规则等。第二是 JavaScript 代码库，也就是由别人写好的程序段，你可以在自己的程序里直接使用，而不必再花时间重写。比如数学函数、计算日历的函数，以及搜索和操作文本的函数。第三是访问浏览器和网页的接口，JavaScript 程序通过这些接口可以在其所在的网页中获得用户输入、响应用户动作（如单击按钮或填写表单）、让浏览器显示不同的内容或者切换到其他网页。

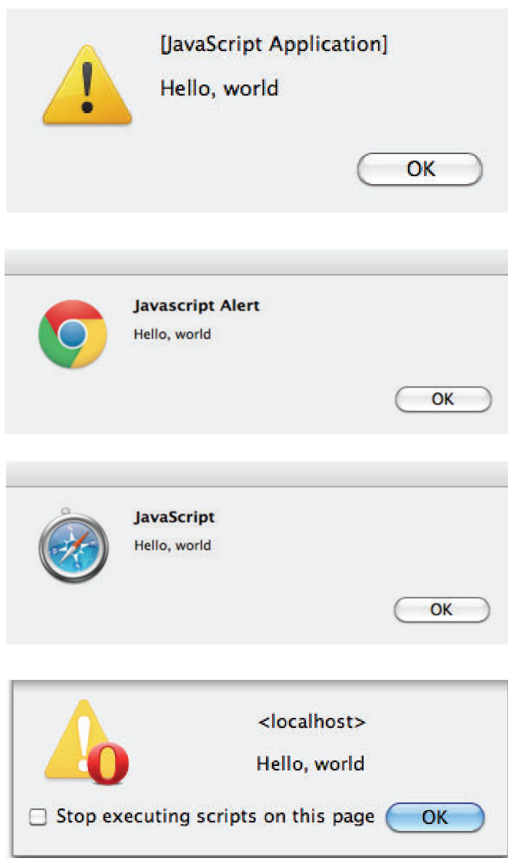
7.2　第一个 JavaScript 程序

下面这个 JavaScript 示例可以说是一个小得不能再小的程序了，它的作用就是在页面加载时弹出一个对话框，显示出“Hello, world”。下面就是完整的 HTML 代码（第 10 章在介绍万维网的时候还会再介绍 HTML），但现在我们只关心其中加粗的那一行 JavaScript 代码，它位于`<script>`和`</script>`标签之间。

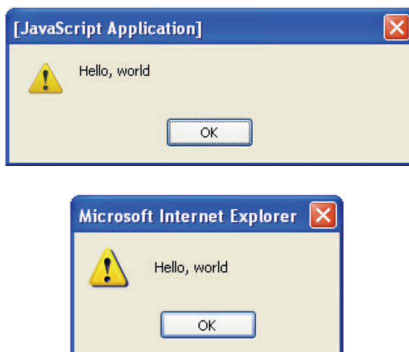
```
<html>
<body>
```

```
        <script>
          alert("Hello, world");
        </script>
    </body>
    </html>
```

把这 7 行代码放到一个名为 hello.html 的文件里，通过浏览器打开它，就会看到下面几个对话框中的一个：



这些图片分别是 Firefox、Chrome、Safari 和 Opera 中对话框的截图（全都在苹果电脑上）。很明显，这些对话框长得都不一样。尽管它们“长相”上的差异无关紧要，但这些差异却很好地说明了不同浏览器的表现可能会不一样。下面再给出 Windows XP 版本的 Firefox 和 Internet Explorer 中相应对话框的截图，仅为比较：



这个程序的 `alert` 函数来自辅助与浏览器交互的 JavaScript 库，调用它会弹出一个对话框，对话框将显示位于引号中的文本。顺便说一句，你在写 JavaScript 程序时，必须使用标准的双引号（"），不要使用所谓的"智能引号"（本书后面会介绍）。这也是讲究语法规则的一个例子。另外，也不要使用 Word 等文字处理程序来生成 HTML 文件，而应该使用记事本或 TextEdit 这样的文本编辑器。在保存程序文件时，要把它保存成扩展名为.html 的纯文本文件（也就是没有任何格式信息的文件）。

学会写这个最简单的程序之后，就可以进一步探索更有意思的编程任务了。从现在起，本书不再给出 HTML 标签，只给出位于`<script>`标签间的 JavaScript 代码。

7.3 第二个 JavaScript 程序

我们的第二个小程序会询问用户的名字，然后再显示一句针对用户的问候语：

```
var username;
username = prompt("What's your name?");
alert("Hello, " + username);
```

这个程序涉及几个新元素和新概念。首先，单词 `var` 添加或者说**声明**了一个**变量**。变量是 RAM 中的一个位置，可以让程序在运行期间存储数据。之所以称它为变量，是因为它的值会随着程序的执行而变化。在高级语言里，声明变量就相当于我们在玩具汇编语言中为一个内存位置起一个名字。打个比方，声明就好比一出戏里的演员表。在这里，我们把这个变量叫做 `username`。当然也可以给它起别的名字，但 `username` 让人一看就知道它在程序中扮演什么角色。

其次，这个程序使用了一个 JavaScript 库函数 `prompt`。`prompt` 与 `alert` 类似，都会弹出一个对话框。但不同的是，`prompt` 能收集用户的输入。用户在对话框中输入的任何内容都会成为 `prompt` 函数中可以使用的值。这个值通过下面这行代码被赋给了变量 `username`：

```
username = prompt("What's your name?");
```

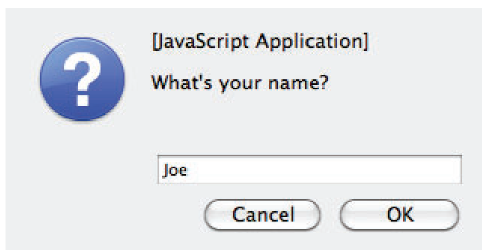
这里等号=的意思是："完成右边的计算，把计算结果保存在左边的变量里。"这个等号也是语义的一个例子。等号执行的操作叫**赋值**。大多数编程语言都使用等号来表示赋值，而没有顾及等号在数学中表示相等的含义。换句话说，这里的=不表示相等，而表示复制值。

最后，`alert` 语句

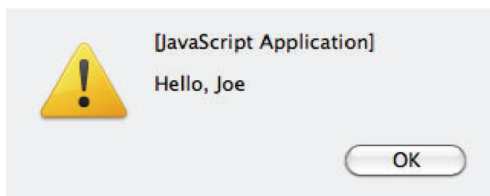
```
alert("Hello, " + username);
```

中使用加号+把单词 Hello（逗号、空格）和用户输入的名字拼接了起来。在这个上下文里，不少人也会感到困惑，因为+一般表示两个数值相加，而非拼接两个字符序列。

运行这个程序，`prompt` 会显示一个对话框，让用户在里面输入内容：



如果你在里面输入"Joe"，然后单击 OK 按钮，会看到下面的结果：



对这个程序进行简单的扩展，可以分别取得用户的姓和名。至于怎么扩展，那方法可就多了，有兴趣的话你可以亲手试一试。注意，如果你在这里输入的是“My name is Joe”，那么问候语也会变成“Hello, My name is Joe”。想让计算机变得更聪明一点？那就得看你的程序怎么写了。

7.4 循环

还记得第 5 章展示的那个 JavaScript 程序吗？那个程序可以把一系列数值累加起来。不用往前翻了，那个程序的代码如下：

```
var num, sum;
sum = 0;
num = prompt("Enter new value, or 0 to end");
while (num != 0) {
    sum = sum + parseInt(num);
    num = prompt("Enter new value, or 0 to end");
}
alert("Sum = " + sum);
```

再提醒一下，这个程序会不断读取用户输入，而在用户输入“0”的时候，会弹出对话框显示之前输入的所有数值之和。这个程序里的一些特性前面刚刚介绍过，比如声明、赋值，还有 `prompt` 函数。其中第一行代码声明了两个后面会用到的变量 `num` 和 `sum`。第二行代码是一个赋值语句，把变量 `sum` 的值设为 `0`。第三行代码把变量 `num` 的值设为用户在对话框中输入的值。

这里真正值得讲的是 `while` 循环，也就是第四行到第七行代码。计算机最擅长一遍又一遍地反复执行一系列指令，而程序员需要想清楚的则是如何通过编程语言来表达这种反复。在玩具语言里，我们添加了 `GOTO` 指令，用于跳转到程序中的另一个位置，而不是顺序执行下一条指令。还添加了 `IFZERO` 指令，用于测试一个条件并根据累加器的值决定是否跳转。

这些概念在大多数高级语言里都有，而且被抽象为一个叫做 `while` **循环**的语句。这个循环语句在反复执行一系列指令时更有规律性，也更加有条理。这个程序里的 `while` 测试了（写在括号中的）一个条件，如果条件为真，则执行花括号`{ ... }`中

所有语句。然后返回，再次测试同一个条件。这个循环一直反复，直至条件为假。此时，接着执行紧跟在右花括号后面的语句。

这个逻辑跟第 3 章使用玩具语言的 IFZERO 和 GOTO 实现的逻辑是不是很像呢？没错，除了不必发明新标签和可以测试任何条件之外，它们是完全一样的。这个程序中测试的是变量 num 是否有一个不等于零的值。因为运算符!=的含义就是“不等于”，这个运算符和 while 语句都是从 C 语言借鉴来的。

我并没有明确指定这个示例程序中数据的类型。但在内部，计算机会帮我们明确区分 123 这样的数值和 Hello 这样的任意字符串。有些语言要求程序员自己谨慎地表达这种区别，另一些语言则试图猜测程序员的意图。JavaScript 差不多就属于后一类，因此有时候明确知道数据类型以及如何处理相应的值是十分必要的。比如 parseInt 函数吧，它可以把文本内容转换成数值，也就是说它的输入数据可以被当成（123 这样的）整数而非三个十进制数字来看待。如果不用 parseInt，那么 prompt 返回的数据就会被当成文本，而+运算符就会把它追加到之前的文本后面。结果将是把用户输入的所有数字逐个拼接起来。这听起来似乎还挺有意思，但却不是我们想要的。

7.5　条件

接下来的这个例子要完成的任务有点不一样，它要从输入的所有数值中找出最大的一个。而这也正是添加另一个控制流语句 if-else 的原因。所有高级语言都有这个语句（可能形式上稍有差异），用于条件判断。实际上，if-else 就是 IFZERO 的通用版本。JavaScript 中的 if-else 语句跟 C 中的一样。

```
var max, num;
max = 0;
num = prompt("Enter new value, or 0 to end");
while (num != 0) {
    if (parseInt(num) > max) {
        max = num;
    }
    num = prompt("Enter new value, or 0 to end");
}
alert("Maximum is " + max);
```

if-else 语句有两种表现形式。一种就是这里所展示的这样，没有 else 子句。此时，只要括号中的条件为真，那么就会执行后面括号{ ... }中的语句。但不管怎样，都会执行紧跟在右花括号后面的语句。另一种形式就是还有一个 else 子句，它也带有一组语句，会在条件为假时执行。无论条件真假，整个 if-else 语句块后面的语句都会执行。

可能你也注意到了，这个示例程序是通过缩进来表示结构的：while 和 if 语句都缩进了。这是一种标准（值得提倡）的做法，能让人一眼就看出 while 和 if 语句都控制着哪些语句。甚至还有一些编程语言要求遵循一致的缩进规则。

把这段程序放到一个网页里就可以测试了。但专业的程序员不必把它放到网页里也能模拟其行为。他们会像计算机一样，认真地推敲每一条语句。如果你也能做到这样（这是确保理解程序的好办法），就可以推断出输入任意值程序都能得出正确的结果。

真的吗？如果输入中包含正数那没问题，但要是输入的都是负数呢？你会发现程序始终会说最大的数是零。

想一想这是为什么。这个程序把到目前为止发现的最大值保存在变量 max 中（就像找出房间里个子最高的人一样）。为了跟后续的数值进行比较，这个变量必须有一个初始值，因此程序一开始（在用户提供任何值之前）就把它设为零。要是用户真的输入了一个大于零的值固然好，就像输入身高一样。可要是用户输入的都是负值（在录入信用卡账单时有这种可能），程序不会输出最大的负值，而是会输出那个终止输入的值。

这个问题容易解决。本章后面会给出一个方案，不过这倒是一个练习的好机会，你也试试吧。

通过这个例子还可以学到编程的另一个重要方面：测试。测试远不止是随机地向程序抛出几个数值那么简单。好的测试人员会绞尽脑汁地想象程序会在什么情况下出错，想象那些“边缘”或“边界”情形，比如根本没有数据或者被零除。好的测试人员会想到输入都是负值的可能性。但问题是，随着程序越写越大，想象出所有测试用例的难度也越来越大，因为用户可能会以任意次序、在任意时间输入任意值。没有完美的解决方案，这时候认真地设计和实现程序就显得很关键。比如从一开始就在程序里添加检测和比较代码，以便在出现问题时，程序自己就可以第一时间捕获。

7.6　库和接口

JavaScript 作为一种扩展机制在高级 Web 应用中扮演着十分重要的角色。Google Maps 就是一个典型的例子，它提供了一个库和一套 API，于是所有地图操作就可以通过 JavaScript 程序（而不仅仅是鼠标点击）来控制了。任何人都可以编写自己的 JavaScript 程序在谷歌提供的地图上显示信息。这套 API 使用起来很方便，比如下面这段代码（当然还得有几行 HTML）

```
function initialize() {
    var latlong = new google.maps.LatLng
                  (38.89767967065576, -77.03656196594238);
    var opts = {
        zoom: 18,
        center: latlong,
        mapTypeId: google.maps.MapTypeId.HYBRID
    };
    var map = new google.maps.Map(document.getElement
              ById("map"),opts);
    var marker = new google.maps.Marker({
        position: latlong,
        map: map,
        title: "You are here, more or less"
    });
}
```

就可以显示出下面所示的地图，没准本书读者中就有人住在这个叫白宫的地方呢。



第 11 章还会向大家证明，互联网应用发展的趋势是 JavaScript 应用会越来越多，包括地图这种可以编程控制的接口。在被迫公开源代码的环境下，要保护知识产权很难。如果你在使用 JavaScript 编程，就必须自己想办法。任何人都可以通过在网页上单击右键并选择“查看源代码”看到你的源代码。有些 JavaScript 程序经过了混淆处理，可能是开发者有意为之，也可能是为了加快下载速度而被“瘦身”的结果。经过混淆的程序已经非常难破译了，除非碰上那些顽固的死磕分子。

7.7 JavaScript 怎么工作

大家回忆一下第 3 章关于编译器、汇编器和机器指令的内容。JavaScript 程序会以同样的方式被转换成可以执行的形式，但细节方面却有着明显差异。浏览器在遇到网页中的 JavaScript 代码时（比如解析到`<script>`标签时），就会把代码文本移交给 JavaScript 编译器——通常是一个独立的程序或者是浏览器的一个库。编译器处理程序、检测错误，然后将其编译为与“玩具”类似的一个假想机器的汇编语言指令。当然，这套指令系统包含的指令要多得多，而实际上它正是我们上一章讲过的一种虚拟机。这个虚拟机接着会像玩具模拟器一样也运行一个模拟器，执行 JavaScript 程序设定的指令。模拟器与浏览器保持着密切交互，比如用户单击按钮，浏览器马上就会通知模拟器哪个按钮被单击了。在模拟器希望做点什么的时候，比如弹出一个对话框，它就会调用 `alert` 或 `prompt` 让浏览器照着去做。

关于 JavaScript，我们这里只能介绍这么多。要是你觉得意犹未尽，可以买本专门讲 JavaScript 的书看一看，其实网上也有很多免费教程，有些教程还可以让你在线编写代码，实时查看结果。编程这件事儿可能会让人纠结郁闷，也能给人带来极大乐趣。你甚至可以靠写代码过日子，而且还能过得不错。任何人都可以成为程序员，但如果你可以做到大处着眼小处着手那就难得了。更难得的是注重细节、一丝不苟，因为只要你稍有疏忽，写出的程序就可能出错，甚至根本无法运行。

对了，本章前面还给大家留了一道练习题呢。下面是一种可能的解法：

```
num = prompt("Enter new value, or 0 to end");
max = num;
while (num != 0) ...
```

这里把 max 设为用户输入的第一个数值，无论正负，它都是目前为止最大的值。其他代码都不用动，程序就可以处理各种输入（当然，输入的值为零则提前退出），甚至都可以应付用户什么都不输入的情况。如果想让程序响应更妥当，就必须好好地理解和运用 prompt 函数。

软件部分小结

到目前为止，我们已经谈论了不少关于软件的内容。下面就对前几章的要点作个简要总结。

算法。算法就是一系列精确、无歧义的步骤，可以执行某种任务，然后停止。算法描述了不依赖于任何实现的计算过程。这些步骤由定义明确的基本操作或原始操作构成。算法有很多，我们只介绍了基本的搜索和排序算法。

复杂性。算法的复杂性是对算法要执行的工作量的抽象描述。度量的依据是基本操作（如检测数据项、比较数据项），而表述的是计算次数与数据项数的关系。算法的复杂性可以分为几个层次，就我们介绍的几种算法而言，既有对数级算法（数据量加倍，计算次数只加一）也有线性算法（计算次数与数据量成正比，最常见也最容易表达），还有指数级算法（数据量加一，计算次数加倍）。复杂性度量的是最坏情况（实际的问题很可能要简单得多），而且描述的是一种渐近性质（只有数据量很大的时候才适用）。

编程。算法是抽象的，而程序是具体的。程序是让计算机完成一个任务的所有步骤的具体描述。程序必须考虑内存和时间的限制、数值的大小和精度，以及偏激和恶意用户。

编程语言。编程语言是表达所有计算步骤的记号库，人们可以籍此轻松写出代码来，而且代码可以被翻译成计算机最终可以执行的二进制形式。翻译方式有很多种，但最常见的是使用编译器，有时候还要用汇编器，把用 C 等语言编写的程序转换成二进制形式，以便在计算机上运行。不同的处理器有不同的指令集和指令形式，因此编译器

也会有相应的差异。解释器和虚拟机是模拟真正或假想计算机的程序，可以面向它们编译并运行代码。JavaScript 程序就是面向解释器编译运行的。

库。编写一个在真正计算机上运行的程序要牵扯很多细节，涉及很多常用操作。库以及类似的机制可以提供预制的组件，供程序员在编程时使用。有了库，程序员就可以在既有工作成果基础上开展新工作。今天的编程工作通常都是组织既有组件与编写原创代码并重。组件可能是库函数（比如 JavaScript 程序中用到的那些函数），也可能是像 Google Maps 一样的大型系统，或者是其他 Web 服务。然而，从底层来看，它们都是由程序员使用我们介绍过的语言或没介绍过的类似语言指令编写的。

接口。接口或者 API（应用程序编程接口）是提供服务的软件与使用该服务的软件之间的一种约定。库和组件通过 API 提供服务。操作系统通过自身的系统调用接口让硬件看起来更有章可循，而且可以编程控制。

抽象和虚拟化。使用软件可以隐藏实现的细节或者把实现伪装成其他东西，比如虚拟内存、虚拟机和解释器。

Bug。计算机不懂宽容，因此容易犯错的程序员必须写出某种程度上没有错误的程序来。所有大型程序都有 bug，也就是说有时候会不听使唤。某些 bug 仅仅只是惹人讨厌，比如设计得不好，并不像真正的错误那么严重。（“这不是 bug，而是一个功能”是程序员中流行的说法。）而有些 bug 只有在极端情况下或者罕见的情境中才会出现，往往很难再现，更不用说修复了。但有些 bug 确实严重，甚至会威胁到人身安全。随着软件在关键系统中的应用越来越广，对计算设备中软件责任的认定也变得越来越重要。过去那种“买不买由你，一旦售出概不负责”的说辞应该改一改了。对待软件也应该像对待硬件一样，厂商必须尽到保护用户的合理责任。

根据经验，因为程序是基于既有组件构建的，而原有 bug 都会消灭掉，至少从原理上讲，新程序中的错误应该越来越少。然而，与这些进步因素相对的是随着计算机和语言的发展，系统承载的需求将越来越多样，市场和消费者呼唤新功能带来的压力也会越来越大，于是无法避免的隐患也会层出不穷。总之，bug 将成为我们心中永远的痛。

第三部分

通　　信

在我们这个“三足鼎立”的世界里，通信是除硬件、软件之外的第三极。无论从哪方面看，通信才是真正让一切变得有意思的东西（也可理解为“有了通信，这个时代才有意思”）。有了通信，计算机与计算机之间才能对话。这种对话通常都是为我们服务的，但有时候也会为图谋不轨之人提供便利。今天的大多数系统都集硬件、软件和通信于一身，可以说是三位一体。换句话说，本书前面介绍的内容也是不可或缺的。通信也是很多社会问题的根源，它诱发了难解的隐私和安全问题，导致了个人、企业和政府之间的权利之争。

这一部分将简略回顾一下通信的历史背景，讲一讲网络技术。我们会介绍因特网，这个网络的网络承载着全世界计算机与计算机之间相当可观的通信量。接着就是万维网，即 Web（World Wide Web 的简称）。Web 诞生于 1990 年代中期，它把少数人独享的因特网变成了我们每个人每日的必需品。之后，我们会提到因特网的应用，比如电子邮件、电子商务和社交网络，以及相应的威胁和对策。

人类从知道如何记事的那一天起，就拉开了远程通信的帷幕。在此期间，人们使用过各式各样设计精巧的物理装置。随便挑选一件通信器具，都可以引出一个可歌可泣的故事，写成一本书都不在话下。

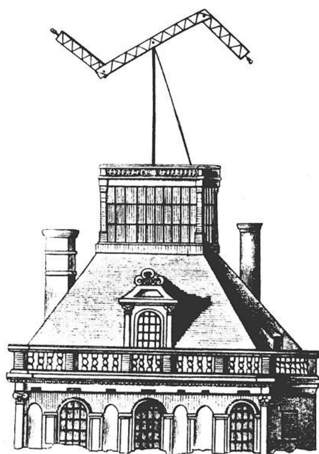
几千年前，人们就曾通过善于长跑的人传递消息。公元前 490 年，费迪皮迪兹（Pheidippides）从马拉松战场奔跑了 42 公里到达雅典，把打败波斯人的胜利消息传递

给了雅典人。不幸的是，跑到雅典之后，他上气不接下气地说完“庆祝吧，我们胜利了”之后就死了（至少传说里是这么讲的）。

希腊历史学家希罗多德（Herodotus ）在他的著作中也提到了同一时期穿越波斯帝国传递消息的骑手。1914 年，希罗多德的话被镌刻在位于纽约第八大道的纽约市邮政总局大楼之上：“无论雨雪、酷热，还是暗夜，都不能阻扰邮差飞快地达成他们送信的使命。”驿马快信（Pony Express）虽然只运营了不到两年（从 1860 年 4 月到 1861 年 10 月），但往返于密苏里州圣约瑟夫和加利福尼亚州萨克拉门托之间 3000 公里邮路上的马背信差，至今仍然美国西部的象征。

可以长距离传递消息的东西包括信号灯和火光、镜子、旗帜、鼓、信鸽，甚至人的声音（英文单词 stentorian 意思是“声音洪亮的”，这个单词就源自希腊单词 stentor，意思是“声音洪亮的人”，他们嗓门非常大，可以向狭窄的山谷对面喊话）。

大约在 1792 年，法国人克劳德・沙普（Claude Chappe）发明了**视觉发报系统**。几乎同时，瑞典人亚伯拉罕・艾地格朗兹（Abraham Edelcrantz）也发明了类似的系统。视觉发报机利用安装在信号塔顶上手工翻转的扇页或摇臂来传递信号。发报员从邻近信号塔接收信号，然后将信号传递到另一方位上的邻近信号塔。到了 1830 年代，这种信号塔网络已经遍及欧洲，并在美国的部分地区使用。信号塔之间相距大约 10 公里，发报速度为每分钟几个字符。据一份文献记载，从里尔向巴黎（230 公里）发送一个字符大约需要 10 分钟。



现代通信系统中的关键问题甚至早在 1790 年代就被注意到了。最重要的，必须有一套共同遵守的协议来表示、交换信息以及检测和排除错误。而且，尽管当时横跨法国发送一条消息只要几个小时，但是想以最快的速度发送信息始终都是个核心问题。另外，安全和隐私问题也出现了。大仲马出版于 1844 年的《基督山伯爵》的第 61 章里，就有化名基督山伯爵的邓蒂斯收买发报员向巴黎发送假消息，从而导致仇人邓格拉斯财务破产的描写。

视觉发报系统有一个致命的问题，就是只能在视线良好的情况下发报，夜晚或恶劣的天气条件下都不能发报。1830 年代，萨缪尔・摩尔斯发明了电报。1840 年代，电报系统在不到十年时间就取代了视觉发报系统。不久，商业电报服务遍及美国各大城市（第一条电报线路 1844 年架设在巴尔的摩和华盛顿之间）。1858 年，第一条穿越大西洋的电报线缆铺设完工。电报系统也让当时的人们经历了从希望到渴望再到失望的过程，就如同后来因特网早期繁荣和 1990 年代后期.com 泡沫破裂一样。财富得而复失，欺诈随处可见。乐观主义者预言世界和平和相互谅解的日子为期不远，而现实的人则认识到尽管情节不同，但历史将再次重演。“这一次不一样”的说法几乎就没有兑现过。

1876 年，亚历山大・格雷厄姆・贝尔（Alexander Graham Bell）就他发明的电话，以先申报几个小时的优势打败伊莱沙・格雷（Elisha Gray）获得美国专利局的专利。尽管事实真相到现在依旧扑朔迷离，但电话在接下来百年的发展，确确实实让人类的沟通和通信有了翻天覆地的变化（当然，同样没有导致世界和平和相互谅解）。电话让人们无需专业知识即可直接通话，而电话公司制定的标准和达成的协议也让世界上几乎任何两部电话之间都能传递语音。

电话的优势在于支持较长时间的稳定通话，但它只能传送语音。一般来说，通话交谈的时间会持续三分钟左右，因此在建立连接上花几秒钟时间不是问题。电话号码是独一无二的一串数字，明确指示地理位置。而用户操作界面则非常简洁，就是一个纯黑色的电话机上面带一个旋转拨号盘（这种电话机现在已非常少见，唯一遗留下来的只剩“拨电话”时的回铃音了）。电话系统的全部智能都在电话网上，用户所要做的只是接听电话、拨打电话，或者让人工接线员帮忙转接。

带旋转拨号盘的电话少见归少见，我一位朋友的母亲现在仍然还在用呢，下面就是朋友发给我的电话照片：



所有这一切的意思很明显，那就是电话系统做到了只关注核心价值：高度可靠和保证服务品质。50 多年来，只要有人一拿起电话，就一定会有一个拨号音（这也是一种回铃音），然后相互通话，对方的声音能够一清二楚地传过来。这个接通的过程一直会持续到双方都挂断才停止。（我对电话系统可能过于偏爱了，毕竟我在 AT&T 所属的贝尔实验室供事长达 30 多年，就算没有接触很核心的东西，也还是了解了不少技术内幕。可不管怎么说，我还是很怀念手机普及之前通话可靠、语音清晰的日子。）

对于电话系统来说，20 世纪的后 25 年是一个快速变化的时期，包括技术、社会和政府政策。1980 年代，传真机的普及让通信方式发生了改变，计算机之间直接通信也变得越来越常见。当时使用的是可以把比特转换成声音（或相反）的调制解调器，俗称“猫”。这种通信方式与传真机的原理相同，都是基于模拟电话网络利用音频来传输数字化的信息。技术的发展让发起呼叫的时间变短了，让能发送的信息量变大了（借助铺设于国内甚至跨越大洋的光纤网络），而且能够将一切都数字化。手机则更为彻底地改变了人们的通信方式，以至于今天的世界完全成了手机的天下。经营有线电话线路的公司每况愈下，后起之秀则层出不穷，各领风骚。

世界范围内的政府政策也发生了巨大变化，曾经是高度管制的行业如今不再受管制。以往，在美国 AT&T 一家独大，其他国家则多由政府部门垄断。现在则实现了开放性的竞争。在美国，AT&T 经历了几次拆分。先是 1984 年被分成长途电话与制造业务（AT&T）和本地电话服务（贝尔电话公司）。之后是 1995 年，新的 AT&T 又自行分为长途通信服务（AT&T）和制造（朗讯）两家公司。AT&T 可谓虎落平阳，最终被它自己在 1984 年分拆出的一家运营商西南贝尔收购。2006 年，朗讯与法国电话设备公司阿尔卡特合并。

今天，传统电话公司仍然在与新兴通信系统殊死搏斗。在这些依托互联网的新兴竞争对手挤压下，他们的收入和市场份额与日俱减。最主要的威胁来自互联网电话，通过互联网发送数字语音是一件轻而易举的事。Skype 之类的服务则更进一步，不仅支持计算机之间的免费通话，而且通过互联网拨打传统电话的收费也不高，至少比现有电话公司的收费便宜得多，尤其是国际通话更加明显。谷歌的服务甚至比这还要便宜（目前在美国和加拿大都免费了），所以说传统电话公司无论如何都在劫难逃。我记得 1990 年代早期，有一位朋友曾跟 AT&T 管理层说过，他说国内长途电话的资费会降到每分钟 1 美分（当时 AT&T 的收费标准大约是每分钟 10 美分）。当时，大家还都笑话他在作无稽之谈呢。

传统电话公司自然不甘败落，他们正使出浑身解数来保护自己的收入，通过技术、法律，甚至政治手段来维护现有的垄断地位。比如，竞争对手要使用你们家的电话线就必须向电话公司付费。这还说得过去，毕竟线路是电话公司铺设到你们家的，人家通过使用费回收成本理所应当。再比如，对于竞争对手通过互联网提供的电话服务（VoIP，即 Voice over IP）或其他竞争性服务，采取限制带宽或其他降速手段。这就不太合理了。

这里涉及一个所谓的**网络中立**（net neutrality）的问题了。除了为确保网络正常运行所要采取的纯技术手段之外，服务提供商有权干扰、限制或者屏蔽网络通信吗？是应该要求电话和有线电视公司也向所有用户提供同样的互联网服务，还是应该允许他们把服务和用户区别开来对待？如果可以区别对待，那依据又是什么？比如说，电话公司有没有权利限制或屏蔽来自其竞争对手（比如 VoIP 公司 Vonage）的流量？有线电视公司有没有权利降低 Netflix 等互联网视频公司的网速？服务提供商有没有权利限制或屏蔽与自己的社会主张或政治主张不同的网站？同样，这些问题大家众说纷纭。

对网络中立问题的看法将从根本上影响互联网未来的发展。而互联网一直以来最得益于（也让我们受益于）其作为一个中立平台的角色，对所有通信都一视同仁，既不干涉，也不限制。

第 8 章

网　　络

这一章，我们介绍几种家庭、办公室或者宿舍里常见的联网技术。我们正是通过这些局域网与互联网连接起来的。

所有通信系统都有一些共性。从最根本处说，它们都是把信息转换成物理表现形式，以便通过某种媒介传输。而在传输目的地，它们再把这些物理形式转换回人们能够理解的形式。

带宽是最基本的一个特性，它描述的是系统传输数据的速度。在供电条件不好、环境很差的情况下，某些系统的带宽可能只有每秒几比特。而与之形成鲜明对比的，则是带宽高达每秒数万亿比特的光纤网络。

等待或**延迟**衡量的是特定信息块通过系统所需要的时间，用卡车满载硬盘穿越整个国家，带宽显然是巨大的，但延迟也是极高的。

抖动，即延迟的可变性，对某些通信系统（特别是语音通信），同样也很重要。

信程指的是某种技术能够在多大地理范围内实现联网。有些网络的范围不过数米，有些网络则可以覆盖全球。

还有一些特性表明发送端口是会向所有接收端口都发送广播消息（跟收音机原理相同），还是会点到点地传输，或者只能在特殊的发送方和接收方之间通信。广播网络固有的问题就是容易被监听，因而遭遇攻击的可能性大，存在安全隐患。为此，必须采取预防措施，以备不测。当然，成本也是要考虑的一个主要因素。

8.1　电话与调制解调器

电话网作为一个覆盖全球的大型网络，从一开始只传送语音，到后来同时传输语音和可观的数据，为人类做出了贡献。大约有近 20 年的时间，人们都是通过电话网把家用计算机接入互联网的。

在家庭中，电话系统传送模拟的声音信号，不传输数据。因此，必须有一种设备来实现数字化信息（比特）和模拟的声音之间的转换，才能利用电话网络传输数据。改变要通过声音信号传输的信息的形式叫**调制**。相反，把这种形式再转换成比特叫**解调**。而能够完成调制和解调功能的设备就叫**调制解调器**（modem）。过去的电话调制解调器是一个价格不菲的大型机柜，后来逐步缩小为一个小芯片，成本也极其低廉。不过，通过有线电话上网的方式在今天已经不常见了，因此配备调制解调器的计算机也越来越少。

使用电话网络传输数据有很多缺点。因为要占用电话线路，所以假如你家里只有一根电话线，那你在上网的同时就不能打电话。不过，对多数人来说，最大的问题还是通过电话线传输数据非常慢。最大带宽不过 56 Kbit/s（即每秒 56 千比特每秒，这里小写的字母 b 代表 bit，一般用大写字母 B 表示 byte），或 6~7 KB/s。那么下载一个 20 KB 的网页，得花 3 秒钟，要是 400 KB/s 的图片呢，差不多要 60 秒。赶上升级软件，轻而易举就得花几个小时。总之，速度太慢。在普遍使用调制解调器上网的年代，大部分时间内的“长途拨号”（实际上也远不到哪儿去）费用也是相当昂贵的。AOL 等提供拨号上网服务的公司，必须提供一系列的市话号码供用户选择，才能让用户免交长途费。即便如此，电话公司也不情愿，因为用户上网占用线路的时间通常都比打电话长，而这些时间又不会产生额外的收入。如前所述，电话网络是为 3 分钟左右的语音通话而非几个小时的数据通信设计的，因此市话一直都按固定标准收费。

8.2　有线和 DSL

电话线传输信号的速度限制是与生俱来的。因为电话只需传输语音，所以有 3 kHz 的带宽就够了。即使再怎么编码、压缩，或者想其他办法，最终都不可能让速度超过 56 Kbit/s。

为此很多人选择了另外两种上网方式，带宽至少是电话线的 10 倍。

首先是使用千家万户都安装的有线电视电缆。这种电缆可以同时传输数百个频道的信号，因此有足够的剩余容量让家庭用户来回传送数据。通常的有线电视网的传输速度都以 Mbit/s 计。来回转换有线信号与比特数据的设备叫**有线调制解调器**。这个名字显然是因为它与电话调制解调器功能相近，当然其速度快多了。

这里所谓的速度快，某种意义上是一种错觉。无论你看不看电视，同样的电视信号都会广播到千家万户。另一方面，虽然大家共享有线电缆，但进入我家的数据就是我的，它不会同时进入你家成为你的数据。换句话说，用户之间没办法共享内容。而数据带宽必须由有线数据的用户共享，如果我用多了，那留给你的就少了。更可能的情况是我们两家的带宽都会变少。值得庆幸的是，我们相互之间倒是不会干扰。通过有线电缆共享上网与航空公司或者酒店有意安排的超额预订非常相似。他们知道预订的人不会全都如约而至，因此超额出售不会出问题。这种策略也同样适用于通信系统。

说到这，又出现了另一个问题。虽然我们实际上都在接收相同的电视信号，但我不希望自己的数据跑到你家去，而你也不希望你的数据跑到我家来。毕竟，这些数据都是个人隐私，比如电子邮件、在线购物订单、银行卡信息，甚至包括自己绝不想让任何人知道的娱乐癖好。这个问题可以通过加密解决，加密可以防止他人偷看我收到或发出的数据。关于加密的话题，我们将在第 10 章讨论。

这里还有一个小插曲。最早的有线网络是单向传输的，即信号会广播到所有家庭。这种网络容易铺设，但用户却不能向有线公司发送信息。有线公司必须解决这个问题，因为视频点播收费和其他服务需要从用户那里接收信息。于是有线网络就变成了双向的，这就为利用有线网络来实现计算机之间的数据通信提供了条件。有时候，“双向通信”会分别使用不同的通道。比如，某些卫星电视系统使用电话线作为上传通道。虽然上网速度慢得让人无法接受，但对于下订单购买电影来说是够用了。而且，上传速度大大低于下载速度也是惯例，所以我们上传图片和视频才会那么慢。

另一种对家庭来说经济适用的联网技术是 DSL（Digital Subscriber Loop，数字用户环路），有时候也叫 ADSL（其中 A 代表 asymmetric，意为“非对称”，因为下载带宽比上传带宽高）。DSL 利用的也是家庭中原有的基础设施——电话线路，而且提供的服务与有线电缆上网相同。但 DSL 与有线电缆相比还是有几个主要的区别。

DSL 在使用电话线发送数据时不会干扰语音信号，这种技术可以让用户打电话和上网冲浪两不误。但 DSL 有距离限制，一般城市或郊区的居民，房子距离本地电话公司的交换机房不超过 5 公里，可以使用 DSL。如果超过这个距离，就没办法用了。

DSL 的另一个优点是非共享。因为它使用电话线，而且也没有其他数据传输服务会同时占线，所以你不会跟自己的邻居共享带宽，你的数据也不会发送到他们家去。DSL 同样需要一个调制解调器，再配合电话公司机房里对应的设备，才能把信号调制成适合在电话线上传输的形式。除此之外，DSL 和有线电缆再没有什么分别了。而且它们的收费标准也差不多，尤其在二者有市场竞争的地区。

这些系统的容量由硬件决定，到了一定的限度之后，再为用户提供更大的容量就不必再购置新设备或增加新投入了。为此，供应商可以玩一个很讨巧的游戏，他答应你只要多掏钱就可以享用更高的带宽。而实际上，他收了钱之后，什么设备也不用添置，只消改动数据库里的一个字段，把你的带宽设置加大就行了。

这种系统都有这个共同的特点，即任何时候都有可用的资源。你想打电话就打电话，想上网就上网。

在技术不断进步的今天，很多地方已经实现了光纤入户，告别了同轴电缆和铜线上网时代。与同类技术相比，光纤传输数据的速度要快得多。光纤中的信号是以光脉冲形式传输的，传输通道是一条极细又极纯净的玻璃纤维，损耗很低。光纤信号在没有中继器的条件下可以传输数公里远。1990 年代早期，我参加了一个“光纤入户”的试验性研究项目，因此我家有 10 年时间在使用（据说是）160 Mbit/s 的光纤上网。虽然我因此有了雄厚的吹牛资本，但可惜如此之高的带宽在当时却没有什么用武之地。

8.3　局域网和以太网

电话、有线电缆和 DSL 等联网技术都可以把计算机连接到一个大型系统，但通常会有一定的距离限制。回顾历史，另外一个网络发展的分支——以太网，最终成为了今天最常用的系统。

1960 年代末至 1970 年代初，施乐公司的帕洛阿尔托研究中心（PARC，Palo Alto Research Center）开发出一台个人计算机，取名“阿尔托”。这台计算机当时只是一个实验用具，但后来却引发了诸多领域的一系列革命。阿尔托拥有世界上第一套窗口系统，配有第一台位图显示器（不局限于显示字符）和第一台激光打印机。尽管相对于今天的个人计算机而言，阿尔托的价格非常昂贵，但 PARC 的研究人员却人手一台。

于是，怎么把所有阿尔托连到一起共享资源（比如打印机）就成了一个问题。而解决方案就是 1970 年代早期由鲍伯・梅特卡夫（Bob Metcalfe）和大卫・博格斯（David Boggs）发明的一种联网技术，叫做**以太网**（Ethernet）。以太网可以在通过同轴电缆相连的计算机之间传送信号。从外观上看，当时的同轴电缆与今天的有线电视使用的同轴电缆很相近。而信号则是基于强度和极性编码比特值的脉冲电压。最简单的情况就是用正电压表示比特 1，用负电压表示比特 0。每台计算机都通过一个设备连接到以太网，而且各有一个独一无二的数字标识符。某台计算机要向另一台计算机发送消息时，它会先通过监听信号来确定没有其他计算机正在向同一台计算机发送消息。然后，它把消息加上接收方的数字标识符广播到电缆中。电缆上连接的所有计算机都可以监听到这条消息，但只有指定的那台计算机才会读取和处理该消息。

每台以太网设备都有一个 48 位的数字标识符，这个标识符独一无二，叫做（以太网）地址。因此，以太网最多可以连接 2^{48}（约为 2.8×10^{14}）个设备。你的计算机也有以太网地址，这些地址通常会印刷在机器底部。当然，在 Windows 中使用 `ipconfig`，在 Mac 上使用 `ifconfig` 也可以显示这些地址。而且经常会有两个地址，一个是有线网卡地址，一个是无线网卡地址。以太网地址都是以十六进制数字表示的，而且两位数字表示一个字节，因此总共是 12 位十六进制数字。比如，我的笔记本电脑的以太网地址就是 `00096BD0E705`，这肯定跟你计算机的地址不一样。

了解了前面讨论的有线网络，也就不难想象以太网也存在同样的两个问题：隐私和资源争用。

资源争用可以借助一个很巧妙的办法来处理：如果某个网络接口在发消息时检测到其他接口正在发，它会先停下来，等一小会儿，然后再发。如果等待时间是随机的，而且会随着尝试次数逐渐加长，那么最终所有消息都会发出去。

隐私在最初的时候并不是问题，毕竟所有人都是同一家公司的员工，还都在同一栋楼里工作。不过到了今天，隐私确实是问题了。因为把以太网的接口设定为“混乱模式”后，就可以读取网络上所有消息的内容（不限于发给它的内容），所以很容易就会与一些重要的个人信息（如未经加密的密码）不期而遇。这种被称为“嗅探”的行为曾经是大学宿舍里使用以太网所面临的主要安全问题。好在，可以通过对电缆中的所有内容加密来解决问题，尽管说服人们使用加密软件并非易事。

想体验一下“嗅探”的感觉？好，推荐你试试一个名叫 Wireshark 的免费软件，它可以显示以太网（包括无线网）流量的各种信息。在发现学生们眼睛只盯着自己的笔记本电脑而不看我的时候，我偶尔会在课堂上播放 Wireshark 的演示。演示虽短，但效果很好，学生们明显会被吸引过来。Firefox 有一个叫 Firesheep 的插件更牛，不仅可以显示谁在访问什么站点，而且还可以让你轻松地伪装成其中的某个用户。Firesheep 可以监控开放（未加密）的网络，从 Facebook、Twitter 及其他站点收集认证信息，用以冒充真实的用户。随便找一个人流密集的地方，几分钟内就能捕获一二十个连接，然后只须点一下按钮就可以悄无声息地伪装成其中某个人。（警告：这样做很可能给你自己惹上麻烦。）

以太网中的信息以包的形式传输。顾名思义，**包**（packet）就是包装比特或字节信息的一种容器，其中信息的格式经过了精确的定义，以便发送时打包，接收时拆包。最形象的比喻就是把包想象成一个信封（或者明信片），上面写着寄信人地址、收信人地址、内容摘要以及其他信息，而且格式都是标准的。就跟联邦快递取货送货时使用的包裹一样。

包的格式因网络不同而异。拿以太网来说，每个包最小 64 字节，最大 1518 字节。其中有 6 个字节用于保存源地址和目标地址，还有其他信息。因此每个包最多会有 1500 字节用于保存数据。

源地址	目标地址	数据长度	数据 （48 ~ 1500 字节）	错误校验

以太网出人意料地获得了巨大成功。它最早被集成到了商业产品中（还不是通过施乐，而是通过梅特卡夫创办的 3COM 公司），随着时间的推移，数不清的制造商卖出的以

太网设备已达数十亿台。早期以太网设备的带宽是 3 Mbit/s，而今天 100 Mbit/s 乃至 10 Gbit/s 带宽的设备都已经随处可见。与调制解调器一样，第一代以太网设备既笨重又昂贵，现如今的以太网卡呢，不过是一小块便宜的芯片而已。

最早的同轴电缆已经被一种 8 芯电缆取代，电缆两头分别是一个常见的 8 针 RJ45 水晶头。这就是我们今天常说的网线。通过网线，可以把设备连接到集线器或交换机，后者会向相连的其他网络发送收到的数据。你的计算机多半都会有一个网口，可以插上标准的水晶头。这种网口也常见于无线基站、电缆调制解调器和 DSL 调制解调器等模拟以太网的设备。



以太网的传输距离也是有限制的，通常在几百米左右。通过硬件（中继设备）连接不同的网段并转发数据包，可以扩大传输范围。智能中继设备可以自动判断主机位置，只将包发送到它应该去的地方。所有这些细节对最终用户都是隐藏的，只有系统管理员在排除故障时才会关心。

8.4 无线网络

以太网有一个明显的缺点——离不开网线。这些网线要么在墙壁里斗折蛇行，要么在地板下匍匐蜿蜒。有时候（我说的是我个人的经历），网线还会穿墙越室，顺梯而下，赫然招摇于餐厅、厨房和客厅之中。连接到以太网的计算机一般都放在几个固定的地方，不能轻易移动。如果你偏爱把笔记本搁在大腿上，那这些网线可真够烦人的。

无线网络同时解决了上网和移动的问题。无线网络（当然不用网线）通过无线电波传输数据，只要信号强度足够，在任何地方都可以通信。无线信号覆盖的范围从几十米到几百米不等。与电视遥控器使用的红外线不同，无线信号不一定非要直线传播。但是，金属物（比如墙壁和隔断）和混凝土结构（如楼板）会干扰无线信号，导致其实际覆盖的范围要远小于在空旷环境下所能覆盖的范围。

从技术角度讲，无线网络利用电磁波传送信号。电磁波是特定频率的电波，其振动频率以 Hz 来衡量（读者可能更熟悉广播电台常用的 MHz 或 GHz，比如北京交通广播电台的频率是 103.9 MHz）。在发送信号之前，首先要通过调制把数据信号附加到载波上。比如，调幅（AM）就是通过改变载波的振幅或强度来传达信息，而调频（FM）的原理则是围绕一个中心值来改变载波的频率。接收器接收到信号的强度与发射器的功率成正比，与到发射器距离的平方成反比。由于存在这种二次方递减的关系，距离发射器的距离增加一倍，接收器接收到的信号强度就只有原来的四分之一。无线电波穿越各种物质时强度都会衰减，物质不同衰减程度也不同，比如说金属就会屏蔽任何电波。高频比低频更容易被吸收，二者在其他方面都一样。

无线联网对可以使用的频率范围——**频段**，以及使用多大的功率发送电波都有严格规定。频段分配始终都是一个有争议的话题，因为各种需求总会发生冲突。频段在美国由 FCC（Federal Communications Commission，联邦通信委员会）等政府机构负责分配，联合国下辖的 ITU（International Telecommunications Union，国际电信联盟）负责制定国际协议。

今天，计算机网络使用的无线标准有一个朗朗上口的名字，叫 IEEE 802.11b/g/n。但民间流行的称呼则是 Wi-Fi（音 wai-fai）。IEEE 指的是 Institute of Electrical and Electronics Engineers，即电气与电子工程师协会。这个协会除了开展其他工作之外，还负责制定一系列电子方面的标准，包括无线通信标准。802.11 是无线通信标准的编号，而这个标准由几个部分组成，其中 802.11b 定义的是 11 Mbit/s 标准，802.11g 定义 54 Mbit/s 标准、802.11n 定义 600 Mbit/s 标准。这些带宽都是名义上的，实用条件下没那么高，大约只有名义带宽的一半。

无线设备可以把数字化信息编码为适合通过无线电波传输的形式。802.11 无线网络中的数据包与以太网中的数据包类似，因此可以用无线连接代替以太网连接。两者的传

输距离也相近，只不过无线网络少了电缆的羁绊。

无线以太网设备发射的电波频率为 2.4~2.5 GHz，某些 802.11 设备的频率会达到 5 GHz。所有无线设备的频率都局限于这一较窄的范围内，冲突的可能性大大增加。更糟的是，有些无线电话、医疗设备，甚至微波炉也跟着凑热闹，同样使用这一频段。有一次我在使用厨房里那台旧笔记本时无线连接突然断了，后来才发现是我用微波炉加热咖啡的缘故。30 秒钟的加热就足以让笔记本断开无线连接。

接下来我给大家简要介绍三种使用最广泛的无线联网技术。首先就是蓝牙，这个名字源自丹麦国王 Harald Bluetooth（约公元 935—985 年）。蓝牙技术是为近距离临时性连接而发明的，使用与 802.11 相同的 2.4 GHz 频段。蓝牙连接的距离是 1 到 100 米，具体取决于功率大小，数据传输速度为 1 ~ 3 Mbit/s。使用蓝牙技术的设备主要包括无线麦克风、耳机、键盘、鼠标、游戏手柄，功率相对较低。

第二种技术是 RFID（radio-frequency identification），即无线射频识别，主要用于电子门禁、各种商品的电子标签、自动收费系统、宠物植入芯片，以及护照等身份证明。RFID 标签其实就是一个小型无线信号收发装置，对外广播身份信息。被动式标签不带电源，通过天线接收到的 RFID 读取器广播的信号来驱动。RFID 系统使用多种不同的频率，比较常见的是 13.56 MHz。RFID 芯片让秘密监视物体和人的行踪成为可能。植入宠物体内的芯片就是一种常见的应用，我家的小猫身上就有一颗。你猜得对，已经有人建议也给人植入这种芯片了。至于动机嘛，就不好说了。

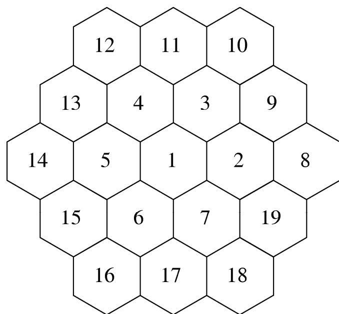
最后一种是 GPS（Global Positioning System，全球定位系统），它是一种重要的单向无线系统，常见于汽车和手机导航系统中。GPS 卫星会广播精确的时间信息，而 GPS 接收器会根据它从三四颗卫星接收到信号的时间来计算自己在地面的位置。然而，GPS 只接收信号不发送信号。以前曾有一个关于 GPS 的误解，认为它能悄悄地跟踪用户。我给大家摘录一段《纽约时报》几年前闹的一个笑话吧：“有些（手机）依靠全球定位系统，也就是 GPS，通过向卫星发送信号来精确地定位用户。”这完全是误解。要想利用 GPS 跟踪用户，必须得有地面系统（比如手机）转发位置信息。正如下一节所要讨论的，手机与基站之间保持密切通信，因而可以（而且确实会）不断地报告你的位置。只不过有了 GPS 接收器之后，它所报告的信息可以更加精确。

8.5 手机

对大多数人来说，最常用的无线设备就是手机了。这种设备曾被叫作“蜂窝电话”或“移动电话”，在 1980 年代还是个稀罕物。如今，地球上半数以上的人都与它须臾不离了。蜂窝电话是本书介绍的所有主题中最值得好好讨论的一个典型，它不仅涉及硬件、软件、通信，还关乎社交、经济、政治，甚至法律。

第一代蜂窝电话是由 AT&T 在 1980 年代早期研制成功的。当时的电话机又大又笨重，而蜂窝电话的广告中则有一个人手提一个装着电池的小箱子，站在一辆携带着天线的汽车旁边。何谓“蜂窝”？因为频段和无线电的覆盖范围都是有限的，因此就要把整个地区划分为蜂窝状的许多小区。可以将每个这样的小区想象为六边形，然后中央有一个**基站**，相邻的小区之间通过基站相连。打电话的时候，手机会与最近的基站通信。当用户移动到另一个小区时，进行中的通话就由原来的小区移交给新小区，但这个切换用户一般觉察不到。

由于接收功率会随着距离的二次方衰减，所以位于既定频段中的频带在不相邻的小区内可以重用，而不会相互干扰。这就是可以高效利用有限频段的秘密所在。大家看下面这幅示意图，1 号小区中的基站与 2 到 7 号小区中的基站不会使用相同的频率，但可以跟 8 到 19 号小区中的基站使用相同的频率，因为与它们之间的距离足以避免干扰了。“蜂窝”中小区的实际形状要取决很多因素，比如天线的辐射图形。这张图只是一种理想化的结果。



实际上，小区覆盖面积的大小并不相同，从几百米到几十公里的都有。具体取决于通

信量、地形、障碍物，等等。

蜂窝手机是常规的电话网络的一部分，只不过连接这个网络不是靠电话线，而是靠基站发射无线电波。蜂窝电话的核心优势就是移动性。手机用户可以长距离（通常也是高速）旅行，中间会穿越很多小区，到了目的地也不一定会收到提醒。比如，我们坐了很长时间的飞机，当飞机降落在目的地之后再打开手机时不会觉得有任何异样。

手机使用的频段很窄，传输信息的能力有限。因为要使用电池，所以打电话时发射的都是低功率无线电波。而且根据法律规定，为了避免与其他无线设备发生干扰，它们的传输功率也受到限制。电池容量越大，待机时间越长，但手机也会更大更沉。这也是一个权衡。

手机在世界的不同地区会使用不同的频带，但一般都在 900 MHz 左右。每个频带被分成多个信道，每次通话时，收发信号各占用一个信道。发送呼叫信号的信道由小区中所有手机共享，在某些系统中这个信道也可以同时用于发送短信和数据。

每个手机都有唯一的识别码（可不是说手机号啊），相当于以太网的地址。启动手机后，它就会广播自己的识别码。距离最近的基站接收到手机信号后，会通过后台系统验证该识别码。随着手机移动，基站实时更新其位置信息，并不断向后台系统报告。如果有人呼叫该手机，后台系统就能通过一直与它保持联系的基站找到它。

手机与基站通信时的信号强度很高。但手机会动态调整功率，在距离基站较近时降低功率。这样不仅可以省电，也可以减少干扰。待机时的耗电量远远比不上一次通话，而这也是为什么待机时间以天为单位，而通话时间以小时为单位的原因。如果手机所在小区信号较弱或根本没有信号，那么它就会因为拼命查找基站而大量耗电。

美国使用了两种完全不同的手机通信技术。AT&T 和 T-Mobile 使用 GSM（Global System for Mobile Communications，全球移动通信系统），这是一种在欧洲使用非常普遍的系统，它把频带分成很窄的信道，在每个信道内依次附加多路通话。GSM 是世界上应用范围最广的系统。Verizon 和 Sprint 使用 CDMA （Code Division Multiple Access，码分多址），这是一种“扩展频段”技术，它把信号扩展到频带之外，但对不同的通话采用不同的编码模式进行调制。这就意味着，虽然所有手机都使用相同的频

带，但大多数情况下通话之间不会发生干扰。

GSM 和 CDMA 都会利用数据压缩来尽可能减少封装信号的比特量。对于通过嘈杂的无线电信道发送数据时无法避免的错误，再添加错误校验来解决问题。

手机带来了一系列难解的非技术问题，频段的分配显然是其中之一。在美国，政府限制每个频带最多只能有两家公司使用指定频率。因此频段是非常稀缺的资源。

手机信号发射塔的位置同样如此。信号发射塔作为户外建筑算不上漂亮，很多地区为此拒绝在自己的地界上搭设这种东西。当然，他们依旧希望得到高品质的手机通话服务。下面这张照片中是“弗兰肯松树”，即勉强能伪装成树的信号发射塔。

电话公司一般会获准在需要的地块搭建信号发射塔，但有时也会经过一番旷日持久的司法程序才能获批。如果它们从那些非纳税机构（比如教堂）租得了架设手机天线的空间，那么发射塔的塔尖经常会是当地最高的结构，就算不想煞风景也不行。



说到社会，手机给我们生活的方方面面带来了翻天覆地的变化。现在的青少年平均每天要发送 50 到 100 条短信（因为提供短信服务所需的成本极低，所以这已经成为电话公司利润极高的收入来源）。iPhone 和 Android 智能手机甚至改变了人们对手机的认知。人们使用手机不再是为了打电话，而是为了其他功能。智能手机还改变了软件开发生态。如今，手机可以上网、收邮件、购物、娱乐和社交，虽然屏幕小了点，但已然成为访问互联网的一股重要力量。没错，笔记本电脑和手机之间存在交叉，因为后者越来越强大，而且便于携带。手机还集成了其他很多设备的功能，包括手表、地址簿、相机、音视频播放器、GPS 导航仪，不胜枚举。

有些手机功能，比如下载电影，需要很大带宽。平板电脑也一样，它不太用于语音传输，主要用在了跟手机一样的其他功能上。随着手机和平板的日益普及，对现有通信设施的压力也会与日俱增。在美国，运营商也开始根据使用量制定阶梯价格和带宽上限，很明显是为了抑制不必要的带宽占用，特别是那些下载大电影的需求。不过这些限制并没有区分忙时闲时，因此起不到鼓励错峰下载的作用。把手机连到笔记本电脑上，让笔记本通过手机上网已经成为可能。运营商不提倡这种使用方式，为此正在谋划着加以限制和额外收费，毕竟这种方式也会占用大量带宽。

8.6 小结

频段是无线联网系统的关键资源，人们对它的需求远远得不到满足。很多政治和经济组织都在激烈争夺频段空间。与此同时，广播公司和电话公司等既得利益集团则在抗拒变化。一种解决方案是有效地利用现有频段。手机通信最早使用的模拟编码技术已被淘汰，当前使用的是更节省带宽的数字系统。现有频段也可能会改换用途，比如 2009 年美国有线电视网经过数字化改造释放出一大块频段空间，成为很多梦寐以求者竞相争夺的目标。当然还可以使用更高的频率，但频率越高覆盖范围越小。

无线网络是广播媒体，任何人都可以监听。加密是保护无线信息和控制访问的唯一途径。事实证明，最初针对 802.11 网络的无线加密标准（WEP，即 Wired Equivalent Privacy，有线等效保密）存在重大缺陷。后来出现的 WPA（Wi-Fi Protected Access，受保护的 Wi-Fi 接入）等加密标准则要安全得多。有些人的无线网络还是开放的，即

没有采取任何加密措施。这样一来，附近的任何人不仅可以监听，还能免费使用其无线服务。“沿街扫描”（war driving）指的是驾车寻找那些有开放网络的地方，享用免费服务。有趣的是，与几年前相比，现在的开放网络似乎少多了，或许是人们已经意识到被窃听和搭便车的风险。

咖啡店、机场等场所的免费 Wi-Fi 服务反而越来越多。咖啡店希望顾客能在自家门店里使用笔记本来消磨时间（顺便也消费几杯昂贵的咖啡）。要是不加密的话，在这些网络上传输的信息也将是公开的，并非所有服务器都会自动加密。当然，知道了有 Firesheep 这种软件存在，那最好就不要使用公共网络发送敏感信息了。

尽管存在这样那样的问题，但无线仍然是未来互联网的趋势。与此同时，有线连接必定也是互联网不可或缺的后台支撑。

第 9 章

互　联　网

前面的章节描述了以太网和无线网等本地网络技术。我们知道，电话系统把全世界的电话机连在了一起，那么如何通过计算机网络把全世界的计算机连接起来呢？怎样才能扩大网络规模，把不同的本地网络连接在一起？如把一幢大楼里的所有局域网都连通，或者用我家的计算机连接到另一个城市里你家的计算机，或者把分别位于加拿大和欧洲的公司网络连成一体。如果底层网络采用了不同的技术，不同的网络又是怎样互通的？当接入的网络和用户越来越多、距离越来越远、所用的设备和技术变化日新月异时，怎样才能从容地扩展网络来满足这些变化的要求？

我们说，“互联网”就是解答以上所有问题的一个答案。实际上，由于它实在太成功了，所以在大多数情况下，它就是**唯一的**答案。

互联网并不是一个巨型网络，更不是一台巨型计算机。它由定义了网络和其中的计算机相互通信规则的标准连接在一起，是一个松散、非结构化、混乱、自组织的网络集合。

如何才能把光纤网、以太网、无线网等不同物理属性的网络连接起来，甚至在它们之间相隔很远时也能连通？我们需要用名字和地址来识别网络和计算机，就像使用电话号码和电话簿那样。我们需要在间接相连的网络之间找到通信的路径，需要就信息采用何种格式传输达成协议，还需要在错误处理、时延、过载等一些不太容易想到的问题上达成协议。没有这些协议，就很难甚至无法进行通信。

所有的网络，尤其是互联网，都需要按照**协议**的约定来处理数据格式、谁先发起请求、后续可以跟随什么样的应答、错误如何处理等方面的问题。在这里，“协议”这个词跟生活中的意思差不多，表示用来跟对方交谈的一组规则。但是，网络协议是基于技术考虑而非社会习俗的，所以它要比最严格的社会结构还精确。

互联网必须满足以下这些不是那么显而易见的规则：互联网的所有接入方都要达成同样的协议和标准，如信息按什么格式组织、在计算机之间怎样交换，如何识别计算机身份并授权，以及错误发生了该如何处理。考虑到既得利益者的影响，网络协议和标准的达成可能会相当复杂：公司要制造设备、售卖服务，专利和技术持有者想到处渔利，政府则想要监视和控制国际和国内的网络信息。

还有稀缺资源的分配问题。在这方面，无线服务的频段就是一个显而易见的例子，网站域名的管理也不能放任自流。由谁来分配这些资源，按什么原则分配？要使用这些有限的资源，应该谁向谁支付，支付什么？资源分配中遇到纠纷谁来裁决？裁决时以哪套司法系统为依据？制定所有这些规则的可以是政府、公司、行业协会，也可以是像联合国下属的国际电联这种名义上的非营利性或中立团体。但无论谁制定规则，最终所有的参与者都要达成一致，照章办事。

显然这些问题都是能解决的，毕竟有先例在：遍布全球的电话系统就把散布在各个国家的设备成功地连接了起来。尽管互联网和电话系统并无本质区别，但互联网比电话系统更新潮、规模更大、变化更快，也更杂乱无章。这个行业对所有人都是开放的，而通信业则是由传统电信公司组成的受控环境，其中大部分公司是由国家垄断或者牢牢控制的。

9.1　互联网概述

在深入了解互联网的技术细节之前，让我们先看看其概貌。互联网初创于 1960 年代，其初衷在于建造一个网络来连接分散在不同地理位置的计算机。由于项目的大部分资金来自美国国防部的高级研究计划署（Advanced Research Project Agency，缩写为 ARPA），最初建成的网络就叫做 ARPANET。1969 年 10 月 29 日，ARPANET 上的第一条消息从加州大学洛杉矶分校发出，到达 550 公里外的斯坦福大学。这一天可以看成互联网的诞生日。

ARPANET 从设计伊始，就具有能处理网络任何局部错误的健壮性，能在网络出现问题时依然成功路由数据。经过多年发展，原始 ARPANET 中的计算机逐步更新换代，所用的技术也推陈出新。网络覆盖范围也从最初只连接美国部分大学计算机科学系和科研机构，到 1990 年代逐步扩大到商界，最终发展成为互联网。

如今的互联网由成千上万个松散连接的独立网络构成，其中的每个网络都连接到另外一个或多个网络。邻近的计算机通过以以太网为主的局域网连接，然后网络和网络再连起来。网络连接采用的设备叫**网关**或**路由器**，其实就是一种专用的计算机，用来把组成信息的数据包从一个网络发送到下一个网络（维基百科上说网关是通用设备，而路由器是专用的，在本书里就不作区分通称为“网关”了）。网关之间互相交换着路由信息，这样它们就至少知道哪些网络与本地网络相连并可以被访问到。

每个网络都可以连接上许多计算机主机（以下简称“主机”），比如家里、办公室里、宿舍里的 PC 和 Mac。家用计算机可以通过无线网卡连接到路由器，然后路由器通过电缆或 DSL 链路连接到**互联网服务提供商**（Internet Service Provider，缩写为 ISP）。办公室的计算机则可用有线网卡与以太网连接。

上一章已经提到，信息在网络之间游弋的时候，被分作称为包（packet）的小块。一个包就是按特定格式组织起来的一串字节。不同设备使用不同的包格式。包的内容中首先是地址信息，用于标明这个包的发送方和接收方；然后是与这个包本身有关的信息，比如包的长度；最后，通常也是最大的部分，是包输送的信息，即有效载荷。

在互联网上，输送数据的包叫做 IP 包（IP 即 Internet Protocol，互联网协议）。所有的 IP 包都是一样的格式。在具体的物理网络上，IP 包通过一个或多个物理包来传输。比如，由于最大的以太网包是 1500 字节，远小于最大 IP 包的 65 000 字节，所以一个较大的 IP 包会被切分成多个较小的以太网包进行传输。

每个 IP 包会经过多个网关，每个网关都把这个包传递给离包的最终目的地更近的下一个网关。一个包在网络中的旅程可能会经过 20 个网关，这些网关很可能是不同国家的公司或研究所拥有和运行的。

要让这一切运转起来，我们需要以下机制。

地址。就像电话号码一样，每台主机都必须有一个辨识身份的地址，才能跟互联网上的其他主机区分开来。这个辨识号码叫做 **IP 地址**，长度为 32 位（4 字节）或 128 位（16 字节）。较短的地址用于互联网协议版本 4（IPv4），长的则用于版本 6（IPv6）。IPv4 已使用多年，现在仍占统治地位，但由于大部分 IPv4 地址都已经分配出去了，网络地址迁移到 IPv6 的进程正日益加快。

IP 地址和以太网地址类似。IPv4 地址通常用其 4 字节的值表示，其中每个字节对应一个十进制数，数与数之间用句点分割，如 128.112.132.86 就是普林斯顿大学网站 `www.princeton.edu` 的 IPv4 地址。这种奇怪的记法叫点分十进制表示法，相比纯十进制或十六进制数值更好记，因而得到广泛的使用。下面是分别用点分十进制、二进制和十六进制表示这个 IP 地址的例子：

十进制	128	.112	.132	.86
二进制	10000000	01110000	10000100	01010110
十六进制	80	70	84	56

IPv6 地址是这个样子：2620:0:1003:100c:9227:e4ff:fee9:05ec，看上去更难理解，所以我们暂时先不讨论。

IP 地址的分配机制是，先由一个中心权威机构把连续的 IP 地址段分配给某个网络的管理员，再由管理员把单个 IP 地址分配给网络里的主机。这样，每台主机就有一个基于它所在网络的独一无二的地址。对台式机来说，IP 地址可能是固定的；但移动设备往往使用动态地址，其 IP 地址至少在设备每次重新连接到互联网时会改变。

名字。由于人们不太擅长记忆无规律的 32 位数字，哪怕写成点分十进制也不好记，因此，要让人们直接访问一台主机，必须给它取个名字才方便。比如，像 `www.stanford.edu` 和 `microsoft.com` 这种常见的名字，我们称之为**域名**。**域名系统**（Domain Name System，缩写为 DNS）用于将名字转换为地址，是互联网基础设施的重要组成部分。

路由。必须有一种机制，能为每个包查找从源地址到目标地址的路径。前文提到的网关就提供了这种功能。网关之间持续交换路由信息，即互联网上网络和设备的相互连接情况，并根据路由信息把收到的每个包转发到下一个离最终目的地更近一些的网关。

协议。最后，为了使信息在不同计算机之间成功复制，必须有一些规则和步骤，用来准确描述上述机制和其他互联网组件是如何协作的。

互联网的核心协议称为 IP，该协议为信息传输定义了统一的传输机制和通用的格式。不同类型的物理网络用各自的底层协议来输送 IP 包。

在 IP 协议之上是传输控制协议（Transmission Control Protocol，缩写为 TCP），该协议利用 IP 协议来提供可靠的传输机制，以便能从源地址向目标地址发送任意长度的字节序列。在 TCP 协议之上，更高层的协议利用 TCP 协议来提供那些我们看起来是“互联网”的服务，如网页浏览、电子邮件、文件共享等。除此之外，互联网还有很多其他协议。例如，IP 地址的动态分配是通过 DHCP 协议（Dynamic Host Configuration Protocol，动态主机配置协议）来处理的。所有这些协议合起来就定义了互联网。

下面我们会详细探讨上述每个话题。

9.2 域名和地址

谁制定规则？谁控制名字和数字地址的分配？谁负责管理互联网？长期以来，互联网由一小群技术专家以松散合作的方式来管理。互联网的大部分核心技术是互联网工程任务组（Internet Engineering Task Force，IETF）开发的。IETF 是一个松散的联盟组织，它设计了构成互联网各要素的运行方式，并将之写成规范的文档。IETF 通过定期召开会议和频繁发布出版物的方式打造互联网的技术规范。这些出版物被称为“征求修正意见书”（Request for Comments，RFC），是互联网的事实规范。RFC 可以在网上找到，迄今为止已有 6000 多个 RFC。但并非所有 RFC 都是一本正经的，不信可以看看 1990 年愚人节应景发布的 RFC-1149——“A Standard for the Transmission of IP Datagrams on Avidan Carriers”（用鸟类运送 IP 数据报的标准）。

管理互联网其他事务的是一个叫互联网名称与数字地址分配机构（Internet Corporation for Assigned Names and Numbers，缩写为 ICANN，其网址为 icann.org）的非营利组织。ICANN 负责互联网的技术协调，包括分配为让互联网正常运行而不能重复的名字和数字地址，如域名、IP 地址和某些协议信息。ICANN 还负责给域名**注册商**授权，好让他

们给个人和机构分配域名。ICANN 最初是美国商务部管辖的一个署，但现在已经是独立的非营利组织，其主要资金来源是对域名注册商和域名注册收取费用。

9.2.1 域名系统

域名系统（Domain Name System，DNS）规定了我们熟悉的分层命名方案，因此就有了 berkeley.edu 和 cnn.com 这样的名字。在域名系统中，.com、.edu 等组织机构代码和.us、.ca 等两个字母的国家代码被称为**顶级域**。顶级域把管理责任和更多名字委托给下级域。例如，普林斯顿大学负责管理 princeton.edu，在这个域名下可为计算机科学系规定子域名 cs.princeton.edu、为古典文学系规定子域名 classics. princeton.edu，计算机系则可进一步规定域名 www.cs.princeton.edu。这种层次式的命名可以一直扩展下去。

域名只表示逻辑结构，并不受任何地理限制。例如，IBM 的分支机构遍布全球，但公司的全部计算机都在 ibm.com 域名下。一台计算机可以为多个域名服务，这在提供网站托管服务的公司里再平常不过了。也可以用多台计算机为同一个域名服务，比如像 Facebook 和 Amazon 这种大型网站。

域名系统没有地理限制会带来很多有趣的结果。比如，南太平洋上位于夏威夷和澳大利亚之间有个群岛国叫图瓦卢，它在域名系统中的国家代码是.tv。图瓦卢把国家代码的使用权租给商业公司，后者则兴高采烈地把.tv 域名卖给客户。如果你要买 news.tv 这种显然有潜在商业价值的域名，可能就要准备掏一大笔钱。反之，kernighan.tv 这种只对姓柯尼汉的人有意义的名字，价格还不到每年 40 美元呢。由于语言上的巧合而受益的其他国家还包括：摩尔多瓦共和国，其.md 域名对医生很有吸引力；意大利，其.it 域名可以用于 play.it 这样的网站。域名通常只能使用 26 个英文字母、数字和连字符。但从 2009 年起，ICANN 批准了一些国际化顶级域名，比如中国除了用.cn 之外也可以用.**中国**。

9.2.2 IP地址

每个网络和每台联网主机都必须有 IP 地址，这样才能互相通信。IPv4 地址是互不重复的 32 位二进制数，在同一时刻任一地址只能给一台主机使用。ICANN 把地址按块分配出去，得到地址块的机构再把它划分成子块分配给下级机构或个人。比如，普林

斯顿大学有两个地址块，分别是 128.122.ddd.ddd 和 140.180.ddd.ddd，其中 ddd 是从 0 到 255 的数字。这里的每个地址块最多允许给 65 536（2^{16}）台主机分配地址，总共大约有 131 000 个可用地址。

这两个地址块没有数值或地理上的关系，这就像 212 是纽约市的长话区号而 213 却是洛杉矶的一样。没有任何理由认为相邻的 IP 地址块代表物理位置相近的计算机，同样也没办法仅靠 IP 地址本身判断地理位置——尽管通常可以从其他信息推断出一个 IP 地址的来源。例如，DNS 支持从 IP 地址到名字的反向查找，可以从 128.112.132.86 查到 www.princeton.edu，于是就可以合理地猜测网站在新泽西州普林斯顿，尽管实际上服务器可能根本就在别的地方。

简单算一下就会发现，IPv4 地址只有大约 43 亿个，甚至还不够地球上每人分一个。因此，按照人类使用的通信服务数量的增长势头，这些 IPv4 地址迟早会被耗光。实际情况比这种"危言耸听"更糟糕，因为 IP 地址是按块划分的，这样用起来就没有理论上那么有效率。想想吧，普林斯顿大学真的会有 13 万 1 千台计算机吗？IPv4 地址耗尽的可能性是真实存在并且迫在眉睫的，大体上在 2012 年就会分配光[①]。

有一种用单个 IP 地址肩扛手挑多台主机的技术，可以让我们在这个灾难面前获得喘息之机。家用无线路由器一般都提供**网络地址转换**服务（Network Address Translation，NAT），即用单个外部 IP 地址为多个内部 IP 地址提供连网服务。使用 NAT 之后，家里所有联网设备从外部网络看都具有相同的 IP 地址，内外地址的双向转换完全由 NAT 设备的硬件和软件搞定。

当互联网地址迁移到使用 128 位地址的 IPv6 之后，这种地址短缺的压力就会消失。IPv6 有 2^{128} 个，也就是大约 10^{38} 个地址，是不会马上耗尽的。

9.2.3 根服务器

DNS 的关键功能是把名字转换为 IP 地址。顶级域的转换工作由一组**根域名服务器**负责，它们知道所有顶级域（比如 `mit.edu`）的 IP 地址。为获取 `www.cs.mit.edu` 的

① 原文如此，实际上 IPv4 地址已经在 2011 年分配完毕，由于原书出版于 2011 年，作者来不及跟踪其最新状态。——译者注

IP 地址，需要先向根服务器查询 `mit.edu` 的 IP 地址，这样就能访问到 MIT 域，然后向 MIT 的域名服务器查询 `cs.mit.edu` 的 IP 地址，从得到的 MIT 计算机系的域名服务器就可以查到 `www.cs.mit.edu` 的 IP 地址。

因此可以说，DNS 使用分而治之的策略来搜索：对顶级域的首次查询把那些在下一步查询中不可能的地址排除掉，后面的每次查询也是如法炮制，这样就能逐级定位到目标主机。

在实际查询中，域名服务器会把近期查找到的名字和地址保存在缓存中。当收到一个新的查询时，如果缓存中有结果，本地域名服务器就会不求助于远程服务器而直接返回了。比如，当我访问 kernighan.com 时，由于十有八九最近没人查询过这个小众域名，本地域名服务器只好从根服务器开始查询。但当我再次访问这个域名时，由于 IP 地址已就近缓存，查询就会快很多。经尝试，首次查询耗时 1/3 秒，过了几秒钟之后再次查询同一个域名，只要首次时间的 1/10，过几分钟再查也是如此。

你可以用 `nslookup` 等工具来亲自体验 DNS 查询。试试运行下面的命令吧：

```
nslookup a.root-servers.net
```

你大概以为只有一台根服务器吧？但对这么关键的系统来说，容忍单点故障的存在显然是个坏主意。因此，共有 13 台根服务器遍布于全世界，其中大约一半在美国。有的根服务器包含了位于天南海北的多台计算机，在功能上和一台计算机一样。它通过某种协议把查询请求分配到这组计算机里最近的那台。根服务器在各种不同硬件上运行不同的软件系统，这样，跟单一环境比起来，遇到系统缺陷或病毒攻击时的系统健壮性就要好很多。尽管如此，根域名服务器一直还是协同攻击的目标。可以想见，在某些特殊条件的组合下，所有根域名服务器也可能会同时宕机。

9.2.4 注册自己的域名

只要你想要的域名还没被别人注册，自己去注册一个域名是很简单的事。ICANN 授权了好几百个域名注册商，你只要挑一个，选择自己想要的域名，付钱，域名就归你了。当然，有些具体限制：域名最多只能有 30 个字符，只能包含字母、数字和连字符。但是只要试一下就知道，对于名字中出现的不端和人身攻击字眼，并没有特别规

定。针对这种情况，大公司和名人只好防患于未然，买下像 bigcorpsucks.com 这种攻击性域名以防挨骂。

光有域名不行，你还需要为自己的网站准备**主机**，也就是存放网站内容以对外提供服务的计算机。这样别人访问你的网站时才可以看到内容。同时还需要找一个**域名服务器**，当别人查询你的域 IP 地址时，好让它告诉人家你主机的 IP 地址。设置域名解析是一个单独的步骤，而某些域名注册商通常也为用户提供域名解析服务，即使自己没有域名服务器，它也会为你寻找其他域名解析服务商提供方便。

竞争导致价格下降。首次域名注册的价格一般为 10~20 美元一年，后期续费也差不多；租用一个空间和流量配置都较低的主机服务的价格每个月大约 5~10 美元。只是用一个通用页面来“停放”[1]域名的服务则是免费的。有些主机服务是免费的；当你只想随意做一个小型网站时，有的主机服务仅象征性收取少量费用；还有的主机服务则给你一段免费的试用期。例如，Google 的域名注册和主机服务每年只要 10 美元。

域名归谁所有？发生纠纷如何解决？别人先注册了 kernighan.com，我该怎么办？答案很简单：先买先得。至于有商业价值的域名，比如 mcdonalds.com 和 apple.com，法庭和 ICANN 的纠纷调解政策则偏向于财大气粗的一方。如果你的名字叫麦当劳或者苹果，基本不可能把域名从这些公司手中抢过来，甚至就算是你先注册的也很难保住。有个真实案例，2003 年，有个叫迈克・罗（Mike Rowe）的加拿大中学生为自己的小软件公司建了个网站 mikerowesoft.com，由于读音跟软件巨头“某软”相似，该公司威胁要为此采取法律行动。这个案子的最后结果是：经调解，迈克・罗换了个域名。

9.3 路由

路由解决从源地址到目标地址的路径寻找问题。在任何网络中，路由都是核心所在。

有些网络使用静态路由表，为所有可能的目标地址提供路径的下一条地址。但在互联网中，由于网络规模太大、动态性太强，使静态路由表难以提供正常的路由。为此，

① 即显示出一个简单页面说明本域名尚未投入使用或准备出售，一般是因为域名持有者尚未准备好网站内容，或者出于投机而购买热门域名。——译者注

互联网网关通过与邻近网关交换信息来刷新自身的路由信息，这样就能保证可能的及所需的路径信息基本跟得上网络的变化。

互联网的庞大规模要求采用分层结构来管理路由信息。在路由系统的最顶层，上万个**自治系统**提供了它们所包含的网络的路由信息。一个自治系统通常也对应于一个大的互联网服务提供商（ISP）。在自治系统内部，路由信息仅进行本地交换，但整个自治系统对外部系统展现统一的路由信息集。

尽管不正式也不严格，路由系统在物理上也存在某种层次结构。用户通过 ISP 接入互联网，ISP 是个公司或组织，它再连接到其他互联网提供商。有的 ISP 规模很小，有的则很大（比如由电话公司或有线电视公司经营的那些 ISP）。有的 ISP 是由公司、学校、政府部门等机构自己运营，是为本机构提供内部服务的；有的 ISP 则对公众提供收费接入服务，如电话公司和有线电视公司。个人通常通过有线网络（在住宅中很常见）或电话接入 ISP，公司和学校则提供以太网或无线连接。

ISP 之间通过网关相互连接。由于主要运营商之间的网络容量巨大，所以各个公司的网络都汇接到运营商的“对等交汇点”，运营商网络之间则互相建立物理连接。这样，就能使来自一个网络的数据高效传送到另一个网络。

有些国家对外连接的网关数量相对较少，这样可以监控和过滤政府认为不宜出现的信息。

你可以用 `traceroute` 程序探测路由信息。Unix 系统（包括 Mac）都提供了这个程序。Windows 上也有，不过名字叫 `tracert`，甚至还有网页版的。下面这个随手给出的例子，显示了从新泽西州普林斯顿到梵蒂冈的网络路径（为适应版面，调整了一些空格）：

```
$ traceroute michael.vatican.va
traceroute to michael.vatican.va (212.77.0.2),
              30 hops max, 40 byte packets
 1 ignition.CS.Princeton.EDU (128.112.155.129) 0.789 ms
 2 csgate.CS.Princeton.EDU (128.112.139.193) 0.756 ms
 3 gigagate1.Princeton.EDU (128.112.12.57) 0.771 ms
 4 vgate1.Princeton.EDU (128.112.12.22) 0.814 ms
 5 local1.princeton.magpi.net (216.27.98.113) 2.819 ms
 6 remote.internet2.magpi.net (216.27.100.54) 4.725 ms
 7 198.32.11.51 (198.32.11.51) 91.898 ms
```

```
 8 so-2-0-0.rt1.fra.de.geant2.net (62.40.112.9) 99.025 ms
 9 as1.rt1.gen.ch.geant2.net (62.40.112.21) 108.320 ms
10 as0.rt1.mil.it.geant2.net (62.40.112.34) 114.343 ms
11 garr-gw.rt1.mil.it.geant2.net (62.40.124.130) 114.505 ms
12 rt1-mi1-rt-mi2.mi2.garr.net (193.206.134.190) 114.691 ms
13 rt-mi2-rt-rm2.rm2.garr.net (193.206.134.230) 124.099 ms
14 83-103-94-182.ip.fastwebnet.it (83.103.94.182) 119.198 ms
15 michael.vatican.va (212.77.0.2) 119.088 ms
```

其中的往返时间表明网络跳越了大西洋。从各个网关名字的神秘编码方式，可以尝试推断网关的位置。我猜测该路由到达意大利之前经过了两三个欧洲国家。有时候，由于网络流量本身和经过的国家等原因，路由到达的地理位置可能令人惊讶，甚至让人完全意想不到。

遗憾的是，出于安全方面的考虑，越来越多的路由站点选择了关闭某些 `traceroute` 运行必需的信息，导致这条命令的实用价值越来越小。

9.4 协议

协议规定了双方互相沟通时遵守的规则：一方是否主动握手，鞠躬多深，谁先从门口走过，在路的哪一侧行驶，等等。虽然有些协议是法律强制规定的，比如在路的哪一边行驶，但生活中的大多数协议都不太正式。

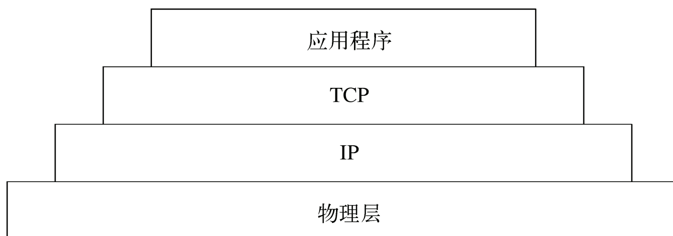
互联网有很多协议，其中最基础的有两个，一是**互联网协议**（Internet Protocol，IP），定义了单个包的格式和传输方式，二是**传输控制协议**（Transmission Control Protocol，TCP），定义了 IP 包如何组合成数据流以及如何连接到服务。两者合起来起就叫 TCP/IP。

由于每种物理网络都使用自己的格式传送 IP 包，网关在接收和转发它们时，必须在特定网络格式和 IP 之间来来回回地转换。

TCP 在 IP 层之上确保可靠通信。这样，用户（实际上是指程序员）就不用考虑分包组包的事，只要面对信息流就可以了。被我们认为属于“互联网”的大部分服务都使用 TCP。

再往上一层是支撑万维网、邮件、文件传输之类服务的应用层协议，它们大多以 TCP 为基础设计。从上面对协议的描述可见，互联网协议是分好几层的，每层都依赖下一

层的服务，并为上一层提供服务。这是第 5 章中提到的软件分层理念的极好例子。下面这张常用的示意图看起来有点像多层的婚礼蛋糕：



和 TCP 处在同一层的协议还有**用户数据报协议**（User Datagram Protocol，UDP）。UDP 比 TCP 简单得多，如果某些数据交换不要求双向流传输，只要高效率分包投递和少量其他特性，那就是 UDP 的用武之地。DNS、流媒体视频、VoIP 和某些在线游戏就使用了 UDP。

9.4.1　互联网协议（IP）

互联网协议（IP）提供的包传递服务是不可靠、无连接的。所谓“无连接”，就是说每个 IP 包都是独立的，和其他 IP 包无关。互联网协议没有状态或记忆力，就是说这个协议一旦把包传给下一个网关，就不再需要保存关于这个包的任何信息。

至于“不可靠”，顾名思义，互联网协议是个“尽力而为”型协议，并不能保证包传送的质量，出错也就出错了。包可能丢失或者损坏，接收到的顺序可能和发送的不一致，也许送达得太快而无法处理，也许送达得太慢而失去作用。当然，实际使用的时候，互联网协议是相当可靠的，但是当包中途丢失或损坏的时候，该协议确实不会尝试修复。这就像是在陌生的地方把明信片丢进邮箱，一般会送达收信人，但可能中途受损，有时会彻底寄丢，有时则比预期时间晚很多到达。互联网协议有一种错误模式倒是在明信片投递上见不到：IP 包可以复制，所以接收方可能会收到多份。

IP 包最大约为 65KB。这样长消息就要拆分成小数据块分别发送，到了远端再组装起来。就像以太网包一样，IP 包也有自己的格式。下图是 IPv4 数据包的格式，IPv6 数据包与之类似，只不过源地址和目标地址都是 128 位的。

版本	类型	头长度	总长度	TTL	源地址	目标地址	错误校验	数据（最大 65 KB）

IP 包中有个很有趣的部分是 TTL（**生存时间**，Time To Live 的缩写）。TTL 是个单字节字段，由包的发送方设置一个初始值，每经过一跳网关就减 1，当减到 0 的时候，就丢弃这个包，并给始发者返回一个报错包。互联网中一次典型的传包过程通常会经过 15 到 20 个网关，所以经过了 255 跳的包显然有问题，很可能是走环路了。TTL 并不能消除环路，但能防止个别包在遇到环路时永远转圈。

9.4.2 传输控制协议（TCP）

在互联网协议簇中，高层协议基于 IP 层的不可靠服务合成可靠的通信，其中最重要的高层协议就是传输控制协议——TCP。TCP 能为用户提供可靠的双向数据流：向一端放入的数据从另一端流出来，延迟很小，出错率很低，仿佛是一条从一头到另一头的直连线缆。

这里不准备讨论 TCP 工作原理的细节（实在一言难尽），但其基本原理倒是相当简单。在 TCP 中，字节流切分成片段，放到 TCP 包也就是所谓的**报文段**里。TCP 报文段不仅包含实际数据，还有控制信息构成的头部，其中包括方便接收方知晓收到的包代表数据流中哪部分的顺序号。通过顺序号，就可以发现丢失的报文段并重传之。TCP 报文段的头部还包括错误检测信息。这样，如果报文段出错，就很容易检测出来。每个 TCP 报文段都放在一个 IP 包里传输。下图展示了 TCP 报文段头部的内容，它们与数据一起封装在 IP 包里：

源端口	目标端口	顺序号	应答	错误检测	其他信息

接收方必须对收到的每个报文段返回确认应答或否认应答。我给你发的每一个报文段，你都要返回一个应答以表明你收到了。如果在适当间隔之后我还没收到应答，那我就认为这个报文段已丢失，然后重新发送。同样，如果你预期会收到某个特定的报文段却没收到，那就得给我发送**否认应答**（比如“未收到 27 号报文段”），这样我就会重新发送。

显然，如果应答报文本身丢失了，情况就会更复杂。TCP 使用若干计时器来检测此类

错误，如果计时器超时，就认为出错。如果某个操作耗时过长，就会启动纠错程序。最终，某个连接可能会因为“超时”而被终止。你也许见过失去响应的网站，那就是遇到了这种情况。这些都是 TCP 协议的一部分。

TCP 协议同样还包含提高传输效率的机制。比如，发送方可以在未收到上个包的应答信息时就继续发送下个包，接收方也可以为接收到的一组包回送一个应答。在通信顺畅的时候，这样做可以降低应答带来的开销。而当网络发生拥塞、开始出现丢包现象时，发送方就迅速回退到低速率，直到慢悠悠地一送一达。

在两台计算机主机之间建立 TCP 连接时，不仅要指定计算机，还要指定计算机上的**端口**。每个端口表示一个独立的会话。端口用两字节（即 16 位）二进制数表示，于是就有 65 536 个可用端口。这样，在理论上每台主机可以同时承载 65 536 个 TCP 会话。

有一百多个众所周知的端口预留给了标准服务。比如，Web 服务器使用 80 端口，邮件服务器使用 25 端口。这样，用浏览器访问 yahoo.com 网站时，浏览器会建立一个到雅虎服务器 80 端口的 TCP 连接，而邮件程序则使用 25 端口来访问雅虎邮箱[①]。源端口和目标端口是 TCP 头部的一部分，头部与数据一起构成 TCP 报文段。

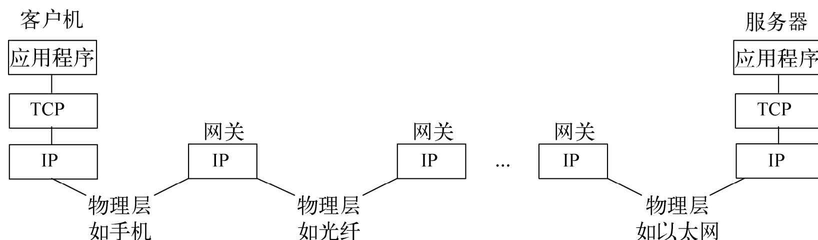
TCP 协议实现细节远比这复杂得多，但基本原理就是这些。TCP 和 IP 最初由文特 · 瑟夫（Vint Cerf）和鲍勃 · 卡恩（Bob Kahn）于 1973 年设计，他们因此一起获得了 2004 年图灵奖。尽管经历了多次改进，但网络规模和通信速度已经增长了多个数量级，TCP/IP 协议还是能基本保持不变，这充分证明最初的设计是相当棒的。如今，TCP/IP 处理了互联网的大部分流量。

9.5 高层协议

TCP 提供双向通信方式，可使数据在两台计算机之间可靠地来回传输。互联网服务和应用程序使用 TCP 作为传输机制，但在完成具体任务时还要使用自己特定的协议。例如，超文本传输协议（HyperText Transfer Protocol，HTTP）就是万维网浏览器和

① 邮件客户端程序访问邮件服务器时，发信和收信使用不同的端口，发信用 25 端口而收信通常用 110 或 143 端口。另外，越来越多的人使用 Webmail，即用浏览器通过邮件服务商的网站收发邮件而不是使用专门的邮件客户端程序，这样使用的端口就和浏览网页一样了。——译者注

服务器使用的非常简单的协议。我在页面中点击亚马逊网站链接的时候，浏览器会打开一个 TCP/IP 连接，连接至服务器 amazon.com 的 80 端口，然后发送一条简短的消息请求某个网页。下图中，左边最顶端的客户端应用程序是浏览器。消息沿着协议链向下层走，跨越互联网（通常是经过更多步传递），到了远端之后回到协议上层，传给相应的服务器应用程序。服务器响应的返回路径不一定和传输客户端请求的路径一样。



亚马逊服务器准备好客户端请求的页面，然后把它和一小段附加数据（比如关于页面编码方式的信息）一起发送回去。浏览器读到返回结果后，据此显示出页面内容。你可以亲自尝试用 Telnet 来摸索这个过程。Telnet 是用来跟别的计算机建立远程登录会话的 TCP 服务，通常使用 23 端口，但也可以指定其他端口。在 Mac 终端或者 Windows 命令提示符窗口里输入下面几行英文：

```
telnet www.cnn.com 80
GET / HTTP/1.0
   [这里再输入一个空行]
```

就会看到返回的 10 万多个字符，浏览器就是根据这样的返回内容来显示页面的。

`GET` 是 HTTP 请求的若干方法之一，“/”的意思是请求服务器上的默认页面，`HTTP/1.0` 则是协议名称和版本号。下一章会详细讲述 HTTP 和万维网。

如果说互联网是信息载体，那我们可以在上面做什么？在本节，我们先看看互联网初期使用的一些最早的程序。这些程序都诞生在 1970 年代初，不过至今仍在使用，主要归功于它们良好的设计和功能。所有这些程序都使用 TCP 来传输字节流。它们都是命令行程序，尽管大多数都很简单，却不是给普通用户而是给那些相对专业的人士使用的。

9.5.1　文件传输协议（FTP）

早期互联网的任务之一是让研究人员能够从别的站点获取各种信息，比如实验数据和测试结果，或者生成和分析这些数据的程序。用来做这件事的工具叫 FTP，即**文件传输协议**（File Transfer Protocol）。

FTP 展示了很多基本理念。其操作过程很简单：打开到一个站点的连接，从那个站点取回文件或者发送文件过去，然后关闭连接。除此之外，FTP 还用一些命令操作远程文件系统里的文件。下面的例子是跟一个 FTP 服务器的对话过程（为了排版紧凑而略做编辑），从中可以看出 FTP 有多么简单。在下例中，我输入的命令用粗斜体表示。

```
$ ftp ftp.cs.princeton.edu
Connected to ftp.cs.princeton.edu.
220 ftp.CS.Princeton.EDU NcFTPd Server ready.
Name (ftp.cs.princeton.edu:bwk): anonymous
331 Guest login ok, send your e-mail address as password.
Password: [我在这里输入 xxx]
230 Logged in anonymously.
Remote system type is UNIX.
ftp> ls （输入 dir 也可以）
150 Data connect accepted from 72.85.133.214:58230
-rw-r-r- 1 ftpuser ftpusers 255749 May 14 15:07 875.pdf
-rw-r-r- 1 ftpuser ftpusers 261940 Jun 18 11:35 876.pdf
-rw-r-r- 1 ftpuser ftpusers 548998 Jun 21 09:31 877.pdf
226 Listing completed.
ftp> get 876.pdf
150 Data connection accepted from 72.85.133.214:58232;
       transfer starting for 876.pdf (261940 bytes).
226 Transfer completed.
261940 bytes received in 00:01 (132.92 KiB/s)
ftp> quit
```

在上面的例子中，尽管服务器不认识客户机的 IP 地址，但仍然允许匿名访问，不检查口令。对于非公共的文件，就要先用某个有权限的帐户登录之后才能访问。

就像早期大多数协议一样，FTP 有个 `help` 命令，你完全可以自己摸索它的用法。

9.5.2　远程登录（Telnet）

第二个早期的工具是 Telnet，该工具用来操作远程机器，其效果就如同直接连在上面

一样。本章前面已经演示过一个 Telnet 的例子。Telnet 接受客户端的键盘输入并将其发送到服务器，仿佛是在服务器上直接按键一样；然后 Telnet 拦住服务器的输出，并发回到客户端。有了 Telnet，就可以登录任何一台联网的计算机，并把它当成本地机器来使用。Telnet 的基本用法很简单，但其附加功能还可以进行文件复制、把键盘输入定向到本机而不是远程等。Telnet 最初的用途是远程登录，但也可用来连接到任何端口，因此也用于对其他协议进行简单测试。下面的例子是用 Telnet 在 Google 上进行搜索：

```
telnet www.google.com 80
GET /search?q=whatever
     [这里再输入一个空行]
```

这个例子输出的结果差不多有 5 万字节，其中大部分是 JavaScript 代码，但仔细看也能在其中分辨出搜索结果。

Telnet 和 FTP 协议一样，都是开放的。如果远程系统能接受没有口令的登录，那就不需要口令。如果远程系统向客户端要口令，Telnet 会以明文形式将客户端的口令发送过去。因此，任何监视数据流的人都能看到口令。现在，除了不讲究安全的场合，Telnet 已经很少用了，原因之一就是它毫无安全性可言。而 Telnet 的继任者 SSH（Secure Shell 的缩写）则因为双向加密了全部通信而得到广泛使用，可以用来安全地交换信息。SSH 使用 22 端口。

9.5.3 简单邮件传输协议（SMTP）

第三个例子是**简单邮件传输协议**（SMTP，Simple Mail Transfer Protocol）。我们通常使用浏览器或者独立程序如 Outlook、Mac Mail 来收发邮件。但就像互联网上的诸多其他应用一样，我们所能看到的只是表面，之下还有若干层，每层都靠各自的程序和协议来支撑。邮件的运行涉及两种基本类型的协议。SMTP 用来在不同系统之间交换邮件。具体步骤是，先建立一条连接到收件人的邮件服务器 25 端口的 TCP/IP 连接，使用 SMTP 协议指明发件人和收件人，然后传送邮件内容。SMTP 是基于文本的协议，理论上可以用 Telnet 连接到邮件服务器的 25 端口观察其运行过程。不过，SMTP 做了足够多的安全限制，因而即使把你自己的计算机当成邮件服务器进行本地操作，也会遇到麻烦。下面的这个实例来自与本机系统的实际会话过程（为排版紧凑而略

做编辑）。这次我给自己发了一封邮件，看上去像是别人发的——实际上就是垃圾邮件[①]：

```
$ telnet localhost 25
Connected to localhost (127.0.0.1).
220 localhost ESMTP Sendmail 8.13.8/8.13.1
HELO localhost
250 localhost Hello localhost [127.0.0.1], pleased to meet you
mail from:liz@royal.gov.uk
250 2.1.0 liz@royal.gov.uk... Sender ok
rcpt to:bwk@princeton.edu
250 2.1.5 bwk@princeton.edu... Recipient ok
data
354 Enter mail, end with "." on a line by itself
Dear Brian --
Would you like to be knighted? Please let me know.
ER
.
250 2.0.0 p4PCJfD4030324 Message accepted for delivery
```

这条纯属瞎扯（或者说至少不现实）的消息就马上送到了我的邮箱：

<No Subject>　　May 25, 2011 8:19 AM

From: liz@royal.gov.uk

```
Dear Brian --
Would you like to be knighted?  Please let me know.
ER
```

由于安全限制，你不太可能在自己的系统上做这件事。如果能的话，你给别人发这样的邮件肯定会惹来麻烦。

SMTP 要求邮件消息是 ASCII 文本。如果要把其他类型的数据转换成文本，或者把多块数据拼成一条邮件消息，那就要按 MIME 标准来。MIME 的意思是多用途互联网邮件扩展（Multipurpose Internet Mail Extensions），实际上它是 SMTP 之外的另一个协议。当需要在邮件里插入照片、音乐等附件时，就要用到 MIME 机制，在 HTTP 中也用得到它。

① 如果读者用的是 Linux 或 Mac OS/X 系统，可以在自己的计算机上做类似实验，但要把里面的收件人域名换成本机的主机名；如果用的是 Windows 系统，由于 Windows 默认未安装 SMTP 服务，建议先远程登录到 Linux 服务器然后再测试。——译者注

虽然 SMTP 也是像 FTP 和 Telnet 那样的端到端协议，但它的 TCP/IP 包从源节点到目的节点常常要经过 15 到 20 跳网关。这就意味着，途中的任何网关都可以检查经过的包，将邮件内容复制下来以从容不迫地审查。而且，SMTP 本身也可以复制邮件内容，邮件系统会跟踪内容和头部的传送。因此，如果不想让发送的邮件内容被别人看到，一定要从一开始就加密。但有一点要记住，加密邮件内容并不会隐藏发件人和收件人的身份。

另外，SMTP 只是把邮件从源主机传送到目的邮件服务器，然后就不管用户怎样收取邮件了。邮件到达目的邮件服务器之后就原地等待，直到收件人取走。广泛使用的邮件接收协议有两个：POP 和 IMAP。

当你把邮件从保存它的邮件服务器系统里移到自己的计算机时，要用到邮局协议（Post Office Protocol，POP）。利用 POP 可以把邮件从服务器上取下来，在自己计算机上保存一份用来阅读，然后把服务器上的那份删掉。

互联网邮件访问协议（Internet Mail Access Protocol，IMAP）则用于与 POP 功能相反的另一种场合：邮件保存在服务器上，你可以从好几个地方访问它。IMAP 确保邮件只存在一个地方，这样任何时候邮箱都是一致状态。由此多个访问者可以同时读取和更新同一个邮箱。因为不需要复制多份消息，也不需要在计算机之间同步，所以，当你需要在好几个地方（比如浏览器和手机）读取邮件的时候，IMAP 就是最佳选择。

像 Gmail 和雅虎邮件这样的“云端”邮件系统也很常见。这些系统底层也是通过 SMTP 传输邮件，客户端也像 IMAP 那样访问邮件。第 11 章将会讲到云计算。

9.5.4 文件共享和点对点协议

1999 年 6 月，美国东北大学一年级新生肖恩・范宁（Shawn Fanning）发布了 Napster 程序，让大家能轻而易举地共享 MP3 格式压缩的音乐。范宁的这事做得可谓恰逢其时：那时的音乐 CD 虽然满大街都是，但价格居高不下，而 MP3 编解码算法已广泛传播开来，个人计算机的运算速度已足以制作并播放 MP3 音乐。网络带宽也足够高，在上面传送歌曲相当快，大学里的宿舍网络就更不用说了。由于范宁的设计和实现做

得很棒，Napster 如星火燎原般传播开来。1999 年中期，他组建了一家公司来经营这项服务，据称最火的时候有 8000 万用户。但同年晚些时候，Napster 就遭遇了第一起诉讼，被指控大范围偷窃受版权保护的音乐。2001 年中期，法院判决 Napster 关闭其业务。两年时间，从白手起家，到 8000 万用户，再到一无所有，这正是后来的流行语“互联网时代”的生动写照。现在，原始的 Napster 已经不复存在，但公司经过重建，名字仍然保留，靠收费的在线音乐谋生[①]。

使用 Napster 的方法是首先下载 Napster 客户端并安装到自己计算机上，然后指定一个文件夹以共享其中的文件。随后客户端登录到 Napster 服务器，把想共享给别人的文件的**文件名**传上去。Napster 维护一个中心目录，里面是可以获取的文件名。这个中心目录一直保持更新，当有新客户端登录上来时，就把它们共享的文件名加进来；现有客户端对系统检测没有响应时，就从列表中去除它们的文件名。

当用户在中心服务器搜索歌名或歌手时，Napster 返回一个列表，里面列出在线而且愿意共享这些文件的其他人。当用户选择了某个共享者时，Naspter 就为双方安排联系：提供 IP 地址和端口号，让用户计算机上的客户端连过去取文件——有点像红娘牵线。供需双方都把状态报告给 Napster，但中心服务器并“不参与”文件共享的过程，从头到尾不碰音乐文件“一个手指头”。

我们已经习惯了客户机–服务器模型，比如浏览器（客户机）从网站（服务器）请求页面。作为一个实例，Napster 为我们展示了另外一种模型：中心服务器列出了现在可共享的音乐，但音乐文件本身还是存储在用户自己的计算机上；当文件传输时，文件直接从一个 Naspter 用户传到另一个用户，并不经过中心系统。这样的组织方式就叫**点对点**，共享者就是其中的对等点（peer）。因为音乐文件本身只存在于对等点计算机上，从来不在中心服务器上，所以 Napster 希望据此规避版权问题，但这些看似合法的技术细节并未得到法庭认可。

Napster 协议使用了 TCP/IP，所以它实际上和 HTTP、SMTP 是同一层的协议。Napster

① 2008 年消费电子产品零售商百思买公司买下了 Napster，保留了其业务。2011 年 10 月，在线音乐服务公司 Rhapsody 公司从百思买收购 Napster，并将其业务和用户合并到 Rhapsody，至此 Naspter 彻底关闭，只保留了网站首页已转跳到 Rhapsody 的服务页面。本书英文原版成书时未来得及跟踪此动向。——译者注

是个简单的系统，范宁的成功是在互联网基础设施、TCP/IP、MP3 和构建图形用户界面的工具这些条件都具备的情况下吹起的一场东风。这并没有贬低范宁工作的意思，因为 Napster 确实是个很精巧的系统。

之后文件共享就流行起来，在 Napster 停业之前就兴起了其他一些服务，有的甚至用来共享并无版权保护的资料。这些服务在几个关键方面与 Napster 有所不同。首先，Napster 系统的脆弱之处在于它把当前可下载的文件列表保存在中心服务器；这样，只要关掉服务器，服务就荡然无存。其次，Napster 公司开在加州，就要受美国法律管辖。之后的文件共享程序就没有这些限制，大多数都在美国境外运作；更重要的是，这些系统用了分布式的目录方案，将可共享文件的信息传播到多台计算机上，只关掉一部分机器并不能停掉服务。同时，这些服务也开始共享其他资料，除了音乐之外还有电影、软件和色情作品，从而在更多的圈子里促进了对文件共享服务的需求。

像 Kazaa 这样的文件共享程序还带了**广告软件**，在客户机运行时会显示广告。这样会在一定程度上影响用户分享文件的热情，不过并不严重。这里有个真实的冷笑话：当有人复制了 Kazaa 的代码，剥离掉其中的广告软件，以“Kazaa 精简版”之名放出来让人用时，Kazaa 的发行者们还抗议了这种侵权行为。

那几年，还有个与 Kazaa 类似的重要的文件共享客户端是 Grokster.com 公司提供的。Grokster 声称它提供的这个程序只用于合法用途，凡使用它产生的任何侵权行为，都是别人的责任。后来，Grokster 遭到一些媒体公司的起诉。2005 年，美国最高法院对该案的最终判决是支持版权持有者。法庭裁定，Grokster 对侵权行为负有间接责任，公司随即停止了业务。现在，网站 grokster.com 还在，值得去看一下，可以看到对侵权后果的预警。[①]

另一个流行的文件共享网站 LimeWire，在 2010 年遭法院命令停业，也显示了类似的警告。

当今的大多数文件共享，不论合法的还是非法的，都使用了一种叫 BitTorrent 的点对

① 该网站会显示出最高法院对此案的判决结果，并在页面下部提示“你的 IP 地址 xx.xx.xx.xx 已经记录在日志，不要以为你不会被抓到，你不是匿名用户”。
——译者注

点协议[1]。这个协议是布莱姆·科亨（Bram Cohen）于 2001 年设计的，特别适用于共享很大的热门文件比如电影和电视节目，因为每个用 BitTorrent 下载文件的站点也必须同时上载一部分文件给那些需要的人。BitTorrent 通过搜寻和 Kazaa 类似的分布式目录来找到想要的文件，并用一个很小的“torrent 文件”来标记传输过程的踪迹，其中维护了谁上载和接收了哪些文件分块的记录。BitTorrent 用户很容易被查到，因为协议规定了下载者也必须上载，这样就可以找到那些把自称受版权保护的资料共享出来的行为。

文件共享是研究本书所涉及各种问题的绝佳案例，它综合了本书几乎所有重要话题。在硬件方面，涉及声音和网络连接的外设，都是 1990 年代早期尚未出现或非常昂贵的设备；用数字形式来表示模拟信息是 CD 和 MP3 的核心所在；软件则包括应用程序和操作系统，客户机–服务器和点对点模型，目录和本机、远程机上的文件，用户界面，当然还有搜索、排序和压缩的算法；通信部分使用互联网和浏览器，要关心带宽、压缩、错误检测。当你让别人在你的计算机上运行他们的程序时，你的安全和隐私就受到了侵犯。

9.5.5 关于版权的题外话

美国的娱乐产业通过美国唱片业协会（Recording Industry Association of America，RIAA）、美国电影协会（Motion Picture Association of America，MPAA）等行会组织对共享有版权资料的行为积极追讨法律赔偿。他们指认了众多所谓的侵权者，提出诉讼或威胁要诉诸公堂。他们热衷于四处游说，试图通过立法来界定共享行为的非法性质。虽然合理付费后再向民众分发高品质音乐似乎有利可图，但接下来事态何去何从仍不明朗。苹果的 iTunes 商店则是这方面的一个成功例子。

在美国，有关数字版权问题的最主要法律是 1998 年的《数字千年版权法案》（Digital Millennium Copyright Act，DMCA）。该法案规定，包括在互联网上散发有版权的资料在内，任何规避数字媒体版权保护技术的行为都是违法的。其他国家也有类似法律。

---

① 作者尚未提及另一个广泛使用的点对点文件共享系统 eDonkey 网络，客户端为 eDonkey2000（电驴）/eMule（电骡）。BitTorrent 的强项是热门资源分享，而在 eDonkey 网络上寻找偏门/古旧资源的成功率高。——译者注

DMCA 是娱乐产业遭遇侵权之后使用的法律武器，但 DMCA 为互联网服务供应商提供了一条“避风港”条款：如果版权持有人通知 ISP 其用户在散发侵权资料，ISP 因此要求侵权人删除版权资料，ISP 不需要对侵权行为负责。避风港条款在大学校园网里很重要，在这里大学就是学校师生的 ISP。因此每所大学都有一间处理侵权举报的办公室，并且在这里张贴了像这样的显眼链接：

根据公法 105-304 号《数字千年版权法案》（DMCA），哈佛大学指派专人接收发生于 harvard.edu 域内的侵权指认通知。若您确认贵方的作品受到侵权，请通知我校指派的以下专人。

DMCA 也常常在势均力敌的法律纷争中为双方共同援引。2007 年，美国影视业巨头 Viacom 起诉 Google 的 YouTube 视频网站提供了该公司版权所有的内容，要求赔偿 10 亿美元。Viacom 认为 DMCA 并未允许以“批发”的形式大规模盗取有版权资料。Google 在自辩中则提到，对于 DMCA 所指的侵权作品下架通知，它总是会做出得体的响应，但 Viacom 却从没有以恰当的方式通知它。2010 年 6 月，法庭判决 Google 胜诉。

再后来又有了新动向，美国政府也在加强版权保护这件事上凑热闹。在采取了一系列有争议的行动后，美国国土安全部下属的移民与归化局（ICE）没收了上百个域名，理由是这些域名被指认存放了盗版资料，或者陈列赝品待售。域名持有者可以上诉以夺回其域名所有权，但如果他们真的站不住脚，到美国之外的域名注册商申请新域名也是轻而易举的事。

9.5.6 高层协议总结

本节提到的这些高层协议的共同之处，就是它们规定的都是如何在计算机程序之间搬运信息，即把互联网作为一个傻瓜网络，从一台计算机到另一台高效地复制字节流，但不去解释或处理这些数据。这是互联网的重要特性：从数据原样传送的意义来讲，互联网就是个“傻瓜”。说得不那么难听点（同时也是**端对端原理**）：连同收发数据的程序在内的各个端点是智能的。与此相反的是传统的电话网络，其所有智能都体现在网络上；而端点设备，比如老式电话机，真的是傻瓜，除了连接到网络上传递声音之外，什么多余的功能都没有。

就是这种“傻瓜网络”却激发出了很高的创造力，因为这种模式意味着任何人只要有好的想法都可以创建出灵巧的端点设备或软件，网络可以为它们传送数据。反之，等待电话或有线电视公司来实现或支持类似的好想法恐怕就不靠谱了。不难预料，这些运营商只乐于加强控制，这一点在日新月异的移动领域表现尤为突出。但与此同时，虽然运营商竭力想控制，大多数革新仍然从别处不请自来。

像 iPhone 和 Android 这样的智能手机根本就是通过电话网络而不是互联网通信的计算机。运营商尽管想从这些电话上运行的服务赚钱，但基本上只能收点数据流量费。早期，大多数手机的数据服务都是每月收取固定费用，但现在（至少在美国），已经逐渐变为按使用量计费的模式了。对于高流量服务，比如下载电影，这样做可能是合理的，但对于发送文本短信这样的服务来说就有点说不过去了，毕竟这种服务占带宽极小，几乎不消耗运营商什么资源。

最后请留意一下早期的协议和程序有多信任使用它们的用户。Telnet 和 FTP 用明文发送口令。FTP 支持匿名传输，而且实际上鼓励这么做。长久以来，SMTP 完全不限制发件人和收件人，任何人发送给任何人的邮件都能转寄。这种“开放转寄”服务是垃圾邮件发送者的天堂——如果你不需要收件人直接回复你，甚至可以伪造发件人地址，如此就很容易制造欺诈和拒绝服务攻击。

简而言之，互联网协议和在其上构建的程序都是针对可信任的个体构成的诚实、协作、善意的团队设计的。当今的互联网已远非如此，所以我们要在各方面对信息安全和用户认证进行亡羊补牢。

9.6 带宽

数据在网络中流动的速度受限于最慢的链路。网络中的很多地方都会减慢流量，链路本身和数据传输路径上计算机对数据的处理过程都是常见的瓶颈。光速也是限制因素之一。信号在真空中的传播速度是每秒 3 亿米（用生活中的尺度说就是每纳秒约 30 厘米），在电路中则慢些，所以就算没有其他延迟，信号从一个地方传到另一个地方也要时间。按照真空中的光速，从美国东海岸到西海岸的 4000 公里要花 13 毫秒时间。相比之下，同样一段路程的互联网，由于经过了十来个路由器，延迟为

40 毫秒。而从美国东海岸到欧洲的互联网延迟是 50 毫秒，到北京是 140 毫秒，到悉尼是 110 毫秒。

生活中，你可能遇到过各种各样的带宽。我的第一只调制解调器每秒钟传输 110 个比特，这速度足以跟上机械打字机。前面章节提到过，现在的调制解调器的速率是 56 Kbit/s。家里的 802.11 无线网络，理论上可以运行在 600 Mbit/s，实际速率则要慢得多。有线以太网的带宽一般是 1 Gbit/s。从家到 ISP 的 DSL 专线或有线宽带连接的速度大概是每秒钟几兆比特，两者的上行速率又比下行速率慢很多。ISP 多半通过光缆连到互联网其他部分，其带宽在理论上有 100 Gbit/s 或更快，但实际则慢得多。

蜂窝电话的带宽跟上面的大部分相比都慢。智能手机的“3G”系统本应该在手机静止时提供 2 Mbit/s 的带宽，运动时的带宽则应为 384 Kbit/s，但 3G 这种不恰当的提法却给不实广告留下了空间。

IP 协议本身并不能保障带宽。实际上，作为“尽力而为”型的服务，它甚至都没有承诺信息一定会送达，更不要说有多么快了。互联网广泛使用缓存来加快数据传输，这在讲域名服务的时候我们都说过了。网页浏览器也缓存信息，所以如果你最近刚访问过某个页面或图像，再次打开它的时候，你所看到的页面或图像很可能来自本机缓存而不是网络。大型的互联网服务器也使用缓存来加快对大量访问者的响应。Akamai 等公司为雅虎等公司提供的内容分发服务，能使内容接收者从距离较近的缓存服务器直接获取内容。搜索引擎也把它们在网页上爬来的海量页面缓存起来，第 10 章会讲到这个话题。

9.7　压缩

数据压缩是更有效利用现有内存和带宽的好方法。压缩的基本思想是避免存储或传输冗余信息。所谓冗余信息，就是那些检索时或在通信链路另一端可以重建、推断出来的信息。压缩的目标是把相同信息编码成更少的比特或位。有些不包含信息的位，可以完全删除；有些位可以从别的位计算出来；有些位对接收者无意义，也可以放心丢弃。

下面以本书的英语文本为例。在所有单词中，字母出现的频率并不相同，e 最常见，其后依次大致是 t、a、o、i 和 n。反过来，z、x 和 q 就非常少见了。用 ASCII（American Standard Code For Information Interchange，美国信息交换标准代码）表示文本时，每个字母占 1 字节也就是 8 位。一种节省 1 位（蚊子腿也是肉啊）的方法，就是只使用 7 位，因为 ASCII 编码的第 8 位（也就是最左一位）永远是 0，不包含信息[①]。同样，可以用更少位来表示那些常见字母，而必要的情况下用较多位表示那些不常见字母。这样，总的位数就可以显著减少。这就跟莫尔斯电码类似：一点表示 e，一划表示 t，但是 q 这种不常用的字母就是“划 – 划 – 点 – 划”。

说得再具体点，就以《傲慢与偏见》英文原文为例吧，该书大约有 97 000 个单词，占 550 000 字节。最常见的字符是单词间的空格，有 91 000 个，其次是 e（55 100 个）、t（36 900 个）、a（33 200 个），而最少的 X（大写）只出现 1 次，第二少的 Z（也是大写）则出现了 3 次。只看小写字母，最少的 j 是 469 次，q 是 509 次，z 和 x 都是 700 次左右。如果我们用两位数（二进制，以下同）来表示空格、e、t 和 a，那就会节省很多存储空间，这样就算 q、x、z、j 和其他不常见字符超过 8 位也没关系。有一种技术叫霍夫曼编码（Huffman coding），可以有条不紊地完成上述操作，为每个字母找到实现最佳压缩效果的编码值。用这种技术，可以把《傲慢与偏见》占用的空间压掉 44%，只剩 310 000 字节，即平均每个字母大约只用 4 位。

如果按大块文字而不是单个字母来压缩，可以得到更好的效果。例如，我们可以根据原始文档的属性选择按单词或词组压缩。在这方面，有好几个算法做得很棒。如广泛使用的 Zip 程序，可以将《傲慢与偏见》占用的空间压掉 64%，只剩 202 000 字节；而我所知的最好的算法（在 Unix 程序 `bzip2` 中），可以将该书压到 145 000 字节，只有原始大小的四分之一。

以上技术都属于**无损压缩**，即压缩过程中不丢失信息，解压后得到的数据和原始数据一模一样。另外一些情况下，并不需要准确重现原始输入，解压后的结果只要大致差不多就足够了（乍一听似乎有点违背直觉）。这时候，使用**有损压缩**可以得到更

① 由于 ASCII 的这个特性，早期的很多串行通信系统，包括电传打字终端、RS-232 串口以至于互联网的某些协议，都为了节省存储空间而只传输 7 位数据。至今还可以在很多互联网协议里看到为了兼容 7 位传输而设计的安全编码方式。——译者注

好的效果。有损压缩最常用于处理要给人看或听的内容。比如压缩数码相机拍出来的照片。人眼分辨不出来非常相近的颜色，所以不必保留实际输入的那么多种颜色，颜色少一点没有任何问题，这样就可以减少编码所用的位数。与此类似，某些难以觉察的细节也可以丢弃，这样处理后的图像尽管没有原始画面那么精密，但眼睛看不出来。细微的亮度变化也是如此。以随处可见的.jpg 图像为例，相应的 JPEG 算法利用前面说的**感知编码**，能把常见图像压缩到十分之一或更小，但看上去也不会明显失真。生成 JPEG 图像的大多数程序都允许控制压缩率，"较高品质"就意味着较低的压缩率。

用于压缩电影和电视节目的 MPEG 系列算法也是按这个道理设计的。在 MPEG 中，单独一帧可以用 JPEG 压缩，还可以把连续的、变化不大的一系列帧压掉一部分。另外，也可以预测画面运动结果，只编码变化部分，甚至把运动的前景从静止的背景里分离出来，减少背景占用的位数。

MP3 是 MPEG 的音频部分，它是一个用来压缩声音的感知编码方案。除了通常的做法外，MP3 利用了人耳无法听到 20 kHz 以上频率（随着年龄增长，这个数值会有所降低）、明亮的声音会掩盖轻柔声音等事实。MP3 编码通常可以把标准 CD 音频压缩到原来的十分之一大小。

手机音频的压缩由于主要集中于人类语音，压缩率可以更高。语音可以比任意音频压缩得更明显，是因为其频率范围较窄，可以从只为个别扬声器建模的单个声道还原出来。知道了特定扬声器的特性，就可以获得更好的压缩效果。

所有压缩算法的思路都是减少或去掉那些不能物尽其用的位串，采用的主要方法包括把出现频率较高的元素编码成短位串、构造频率字典、用数字代替重复内容等。无损压缩能够完美重现原始数据，有损压缩通过丢弃接收者不需要的信息，来达成数据质量和压缩率的折中。

压缩时也可能需要权衡其他因素，如压缩速度和复杂性与解压速度和复杂性。数字电视画面撕裂成块或者声音断断续续，说明解压缩算法碰到了输入错误，可能是因为数据到达的速度不够快。不管是什么压缩算法，有些输入都是无法被压缩的。想象一下用某个算法压缩其前一次压缩得到的结果，你就会知道，任何压缩都是有限度的。

9.8　错误检测和校正

如果压缩是去除冗余信息的过程，那么错误检测和校正就是加入精心控制的冗余信息以便检测错误甚至修正错误的过程。

有些常见数字没有冗余，这样，一旦出现错误就没法检测到。比如，美国的社会保障号码有 9 位，几乎任何 9 位数字都有可能是合法的社会保障号（如果有人要你填社会保障号码但实际上他们又没用时倒好办了，随便编 9 位数字就行）。如果多加几位数字，或者排除掉某些值，就可以检测出错误了。

信用卡和取款卡的卡号有 16 位长，但并非所有的 16 位号码都是有效的卡号[①]。这里使用了 IBM 公司的彼得·卢恩（Peter Luhn）于 1954 年设计的一个**校验和**（checksum）算法，来检测在实际操作中最常见的两种错误：单个数字错误、由于两个数字写错位置而引起的大多数换位错误。

这个算法很简单：从最右一位数开始向左，把每个数字交替乘 1 或 2，如果结果大于 9 就减 9。如果把各位数的计算结果加起来，最后得到的总和能被 10 整除，那这个卡号就是有效卡号。你可以用这个方法测试一下自己的银行卡，或者某些银行广告中出现的卡号，如“4417 1234 5678 9112”。由于这个卡号计算的结果是 69，所以不是真卡号；如果把它的最后一个数字换成 3，那就是有效卡号了。

10 位或 13 位的 ISBN 书号也采用了类似算法的校验和，用来对付同类错误。条形码和美国的邮政编码也使用了校验和。

这些算法都是用于十进制数字的专用算法。应用于二进制位的最简单的错误检测算法是**奇偶校验码**。这种算法为每组二进制位上附加一个**奇偶校验位**，其值的选择要满足如下条件：使该组二进制位中值为 1 的位有偶数个。这样如果出现一位错误，接收者就会看到奇数个 1，从而知道有内容被破坏了。当然，奇偶校验码并不能识别出哪一位有错，也不能检测两位同时出错的情况。

---

① 16 位是较为常见的卡号长度，但按 ISO/IEC 7812 标准规定，可以是 13 到 19 位。美国运通卡是 15 位；中国国内一些大银行的银联借记卡由于持卡人数量庞大，已使用 19 位卡号。——译者注

举个例子，下表的字母栏列出了 ASCII 表中前 6 个大写字母的二进制形式；偶校验栏把没有使用的最左边一位用作校验位，让每个字节里有偶数个 1；而在奇校验栏中，每个字节里有奇数个 1。如果其中任何一位的值发生翻转，得到的字节就通不过奇偶校验，于是错误就会被检测出来。在这个例子中，如果再使用更多的校验位，还可以实现一位纠错。

字母	原始值	偶校验	奇校验
A	01000001	01000001	11000001
B	01000010	01000010	11000010
C	01000011	11000011	01000011
D	01000100	01000100	11000100
E	01000101	11000101	01000101
F	01000110	11000110	01000110
...			

错误检测和校正广泛应用在计算和通信中。虽然针对不同类型的错误要使用不同的算法，但纠错码可以用在任意二进制数据上。比如，内存可以抵御在随机位置上出现的一位错误，CD 和 DVD 用编码防止错误位影响后面的持续播放，手机能抵御短暂的突发噪音。就像压缩一样，错误检测并不是万能的。总会出现这样的情况：一组数据，本来匹配某种合法模式，出错以后却匹配了另一种合法模式。

9.9 小结

互联网背后只有少数几个简单的设计思想，但在大量工程实践的支持下，仅用如此少的机制就实现了非凡的成就。

互联网是基于数据包的网络。在互联网中，信息被封装在一个个标准格式的数据包里发出，动态地在一个巨大的变化无常的网络集合里被路由。这种网络模型和电话系统的电路网络完全不同，后者每次通话都会建立一条专用电路，在概念上可以理解为连接通话双方的私有线路。

互联网给接进来的每台主机分配唯一的 IP 地址，同一个网络内的主机共享同一个 IP 地址前缀。笔记本等移动的主机每次连上网的时候，多半会使用不同的 IP 地址，搬到

不同的地方，其 IP 地址可能会变。域名系统是个巨大的分布式数据库，用来把主机名字转换成 IP 地址，或者反之。

网络之间通过网关连接。网关是一种专用计算机，用来把那些奔赴终点的数据包从一个网络路由到下一个网络。网关用路由协议交换路由信息，这样即使网络拓扑发生改变、网络间的连接此去彼从，网关还是知道如何把一个包发送到离它的终点更近的地方。

互联网依靠协议和标准来生生不息。IP 是互联网的通用机制，是交换信息的通用语言。以太网和无线网等专用硬件技术都封装 IP 包，但 IP 层感觉不到某个具体硬件的工作方式，甚至不知道其存在。TCP 用 IP 建立连接到目标主机指定端口的可靠数据流。更高层协议则用 TCP 创建互联网服务。

协议把系统切分成层，每一层为上一层提供服务，并调用下一层的服务。这种把协议分层部署的模型是互联网运行的基础，是组织和控制复杂性并隐藏不相关实现细节的绝好方法。在分层协议中，每层只关注本层知晓如何完成的任务：硬件网络把字节从网络中的一台计算机搬运到另一台、IP 在互联网中传送数据包、TCP 从 IP 合成可靠的数据流、应用协议用数据流来回发送数据。每层都使用其相邻下层提供的服务，并为其相邻上层提供服务。任何一层都不试图大包大揽。显然，每层展现出来的编程接口都是第 5 章讲到的 API 的绝佳范例。

互联网的隐私和安全是个大麻烦，下面几章会深入讲这些。这就像是入侵者和防御者的军备竞赛，大多数时候入侵者魔高一丈。世界各地散布着人们共用的、无人管制的、各色各样的介质和网站，数据从其中穿行而过，在一路上的任何位置都可能被人出于商业或政治目的记录、审查和阻断。而要在途中控制访问、保护信息，则难于上青天。很多网络技术用到了广播，这使它们面对窃听毫无抵抗之力。攻击以太网需要找到其中的线缆并进行物理连接，但攻击无线网却不需要物理接触就能偷听。

互联网始于 1969 年，TCP/IP 的核心协议在 1973 年亮相。其后，虽然经历了网络规模、流量和应用的爆炸式增长，互联网还是一如既往地开放和高效。这真的是非凡无比的成就。

第 10 章

万 维 网

互联网最外在的一面就是万维网（World Wide Web），也就是我们常说的“上网”的“网”（简称 Web）。虽然平常我们不怎么区分互联网和万维网，但两者其实并不相同。万维网连接着提供信息和请求信息的计算机（提供信息的叫服务器，请求信息的叫客户端，比如我们的个人计算机），它通过互联网建立连接和传送信息，并为互联网支持的其他服务提供人机界面。

像许多伟大的理念一样，万维网在本质上是简单的。除了无处不在、高效、开放的底层网络必不可少外，万维网主要有以下四个组成要素。

首先是 URL（Uniform Resource Locator，统一资源定位符），形如 http://www.amazon.com，用于指定要访问信息的名字以及信息所在位置。

其次是 HTTP（HyperText Transfer Protocol，超文本传输协议），上一章刚刚介绍过这个高层协议。HTTP 的功能很简单，它让客户端能够请求某个 URL，同时让服务器能够返回客户端想要的信息。

然后是 HTML（HyperText Markup Language，超文本标记语言），描述服务器返回信息的格式（或表现形式）。HTML 同样很简单，不需要什么背景知识就能掌握其基本用法。

最后是浏览器，即运行在客户端计算机上的 Firefox、Internet Explorer 等程序，它通过 URL 和 HTTP 向服务器发送请求，然后读取并显示服务器返回的 HTML。

万维网的诞生可以追溯到 1989 年。当时，在日内瓦附近的欧洲核子研究中心（CERN）工作的英国物理学家蒂姆·伯纳斯–李（Tim Berners-Lee），为便于通过互联网共享科学文献和研究结果而设计了一套系统，包括 URL、HTTP 和 HTML，以及一个只能用文本模式查看可用资源的客户端。

这套系统在 1990 年投入使用。说来惭愧，我 1992 年 10 月还亲眼见过有人使用它，可当时并没觉得它有那么好，也根本没想到 6 个月后诞生的第一个图形界面浏览器会改变世界。瞧我这眼光！

世界上第一个图形界面的浏览器 Mosaic，是由伊利诺伊大学的一群学生开发的。Mosaic 的首个版本发布于 1993 年 2 月，很快就大获成功。次年，第一个商业浏览器 Netscape Navigator 面世。Netscape Navigator 是早期的成功者，而那时微软对互联网的蓬勃发展毫无意识。但不久，这个软件巨头还是觉醒了，随后很快推出竞争产品 Internet Explorer（IE）。IE 后来居上，成为最常用的浏览器，市场份额遥遥领先。微软在多个领域的市场统治地位引发了反垄断关注，公司也因此遭到了美国司法部的起诉，其中包括对 IE 的指控。据称微软利用其在操作系统领域的统治地位，将竞争对手 Netscape 排挤出了浏览器市场，详细内容第 5 章介绍过。近年来，随着 Firefox、Safari 和 Chrome 等强有力竞争者的出现，IE 在浏览器领域的主导地位已经被大大削弱。

Web 技术的发展，由万维网联盟（World Wide Web Consortium，简称 W3C，其网站为 w3.org）这个非营利机构控制着（至少是引导者）。W3C 创始人和现任主席伯纳斯-李没想过靠自己的发明赚钱，而是慷慨地提出让所有人免费使用万维网，反倒是很多投身其中的人都托他的福，成了腰缠万贯的大富翁。2004 年，英国女王伊丽莎白二世授予伯纳斯-李爵士勋章。

10.1 万维网如何工作

接下来让我们从 URL 和 HTTP 开始，认真看看万维网背后的技术要件。

假设你打开常用的浏览器，正在看一个简单的网页，页面中有些文字可能是蓝色的并带着下划线。用鼠标点击这些文字，当前页面就会换成蓝色文字指向的新页面。类似这样

相互链接的页面就叫超文本（意思是“不光是文本”）。超文本实际上是个老概念，但浏览器把它推到每个人面前。

假设某个链接的内容是“W3C 主页”，把鼠标移到该链接上，浏览器窗口底部的状态栏就会显示链接指向的 URL，如 http://w3.org，域名之后也许还有其他信息。

点击链接，浏览器就会打开一个到 w3.org 域的 80 端口的 TCP/IP 连接，然后发送 HTTP 请求，获取 URL 中域名后面部分表示的信息。例如，如果链接是 http://w3.org/index.html，那么请求的就是 w3.org 服务器上的 index.html 文件。

收到请求后，w3.org 服务器首先判断接下来该怎么做。如果客户端请求获取的是服务器上的文件，服务器就将该文件发送回去，由客户端（也就是浏览器）显示出来。服务器返回的文件绝大多数都是 HTML 格式的，其中包含实际内容和如何显示这些内容的格式信息。

以上只是最简单的描述，实际情况往往更复杂一些。HTTP 协议规定，浏览器可以在客户端请求中增加若干附加信息。服务器返回的结果中通常也会包括指明数据长度和类型的额外信息。

URL 自身会对信息进行编码。URL 开头的“http”是协议名，最常见的是 HTTP，占了很大比例。如果留意的话，偶尔也会看到其他协议开头的 URL，比如“file”表示信息来自本机（而不是网上）、“ftp”表示使用 FTP 协议传输文件、“https”表示采用经过加密的安全 HTTP 协议（本章稍后介绍该协议）。

接下来，“://”后面是域名，即服务器的名字。域名后面可以跟着斜线（/）和任何一串字符。这些字符串会原样传递给服务器，由服务器决定如何处置。最简单的情况是域名后什么都没有，连斜线也没有。在这种情况下，服务器将返回默认页面，比如 index.html。如果域名后有文件名，就返回其对应文件的内容。文件名之后如果有问号，一般表示问号前面的部分是程序，服务器会运行该程序并把问号后面部分作为参数传入。这就是服务器处理网页表单信息的一种方式。下面可以用 Bing 搜索来验证，比如在浏览器的地址栏里输入下面的 URL：

```
http://www.bing.com/search?q=funny+cat+pictures
```

URL 中的字符必须在某个字符集中，这个字符集里不包含空格和除字母、数字之外的大多数字符。当需要用到字符集之外的字符时，就要先对这些字符进行编码。例如，用加号来表示空格、用前面冠以百分号（%）的十六进制代码表示其他字符。例如下面的 URL 片段

```
8%22%20to%202%27
```

表示

```
8" to 2'
```

因为十六进制 22 表示双引号，27 表示单引号，20 表示空格。

2009 年末，ICANN 批准使用以 Unicode 编码的“国际域名”。迄今为止，国际域名还仅限于顶级域，而且只为少数几个国家提供。以埃及为例，该国除了使用传统的.eg 之外，也可以用阿拉伯文域名

مصر.

显然，这在拉丁字母不通行的地方很有吸引力。

10.2　HTML

服务器返回的绝大多数文件都是 HTML 格式的，其中包含了文本内容和格式信息。HTML 其实相当简单，只要用你常用的文本编辑器就能编写 HTML 网页。（如果你用的是 Word 这样的字处理软件，切记用.txt 纯文本格式保存网页而不要用默认格式。）HTML 文件用**标签**来表示格式信息，标签不仅可以内嵌文件内容，通常还可以标示页面区域的起始和结束位置。一个简单网页的 HTML 代码如下：

```
<html>
  <title> My Page </title>
  <body>
    <h2>A heading</h2>
    <p> A paragraph...
    <p> Another paragraph...
    <img src="news.gif">
    <a href="http://news.google.com">link to Google News</a>
```

```
    <h3>A sub-heading</h3>
    <p> Yet another paragraph
  </body>
</html>
```

这个网页在浏览器里看起来是这样的：

A heading

A paragraph...

Another paragraph ... Google news link to Google News

A sub-heading

Yet another paragraph

如果图像标签`<img>`里的文件无法访问，浏览器可能会在那个位置显示出一张“破损”的替代图片。

在 HTML 中，有些标签是自包含的，比如`<img>`；有些则有始有终，比如`<body>`和`</body>`；还有些标签，如`<p>`，虽然严格定义里要求有闭合标签`</p>`，但在实际中并不需要`</p>`。缩进和换行并非必需，但加上了会让 HTML 代码更易读。

本节讨论的这一点 HTML 恰好够用来揭开网页制作的神秘面纱。基本网页制作技术就是这么简单，只要再花几分钟时间学习，就能做出比上面例子更像样的页面。如果要制作你在商业站点看到的那些精美页面，就要掌握相当多的技能了。但只要学会十来个标签，应该就能胜任大部分纯文字页面，再学十来个，就能做出让一般用户侧目的东西来了。手工创建页面很简单，文字处理程序也有“创建 HTML”的选项。还有很多用来制作专业网页的软件，但只有在你真想学做网页设计的时候才需要。不过，最重要的还是理解网页背后的工作原理。

10.3　表单

HTML 最初只能返回纯文本供浏览器显示。但没过多久，浏览器就得到改进，可以显示图像了，包括简单的 LOGO、GIF 格式的笑脸和 JPEG 格式的照片。此外，用户还可以在浏览器显示的网页里填写表单、按下按钮、弹出新窗口或用新窗口替换当前窗

口。随后，当网络带宽足以支撑快速下载、主机性能可以快速处理图像显示时，声音、动画和电影马上又涌入了网页。

HTTP 协议里有一个从客户端（你的浏览器）向服务器传递信息的机制，叫通用网关接口（Common Gateway Interface，CGI）。只看这个名字，很难想象它能用来传递用户名和密码、查询条件、单选按钮和下拉菜单选项。CGI 机制在 HTML 里用`<form> ... </form>`标签来控制。你可以在`<form>`标签里放入文本输入区、按钮等常见界面元素。如果再加上一个“提交”按钮，按下去就会把表单里的数据发送到服务器，服务器用这些数据作为输入，来运行指定的程序。

下面是一个简单的例子：

```
<form action="http://kernighan.com/cgi-bin/echoname">
     Type your name here:
     <input name="username" type="text">
     <input type="submit">
</form>
```

将上述代码放进网页，就会在浏览器中看到：

Type your name here: John Q Public Submit Query

在网页上输入名字后按下提交按钮，浏览器就把输入的名字（即 `username` 字段的内容）发送到服务器，服务器以它作为输入，运行 echoname 程序。echoname 程序的功能只有一个，就是返回一个嵌有输入名字的页面：

Hello, John Q Public

表单有很多局限，比如：只支持按钮、下拉菜单等少数界面元素；除了编写 JavaScript 代码或把表单数据发送给服务器进行处理外，没办法验证表单数据的正确性；没有对密码输入字段进行任何安全性保护，密码完全以明文的形式发送和存储在日志中。尽管如此，表单仍然是万维网的重要组成部分。

10.4　cookie

HTTP 协议是“无状态”的。“无状态”的意思是，HTTP 服务器不必记住不同客户端

发送的请求信息，只要向客户端返回了请求的页面，它就可丢弃有关这次数据交换的全部记录。

于是，问题就来了：有时候服务器确实需要记住某些东西，如用户已经输入的名字和密码，这样后续的每一次交互就不必让用户反复输入了。怎样才能让 HTTP 记住这些东西呢？难点在于，客户端第一次和第二次访问服务器的时间可能间隔几小时、几星期，也可能访问一次以后再也不会访问，服务器要把信息保留多长时间呢？似乎只能靠瞎猜。

1994 年，Netscape 公司发明了一种叫 cookie 的解决方案。cookie 是在程序之间传递的一小段信息，这个名字虽有卖萌之嫌，但已经为广大程序员接受。向浏览器发送页面时，服务器可以附加若干个浏览器可存储的文本块，每个文本块就是一个 cookie，最大为 4000 字节左右。当浏览器再次访问同一个服务器时，再把 cookie 发送回服务器。没错，服务器就是这样利用客户端的内存来记住之前哪个浏览器曾访问过它的。服务器通常为每个客户端分配一个唯一的识别码，包含在 cookie 里；而和这个识别码相关联的永久信息如登录状态、购物车内容、用户喜好等，则由服务器上的数据库来维护。每当用户再次访问这个网站时，服务器就用 cookie 识别出用户原来之前来过，为其建立或恢复信息。

我通常会禁用 cookie，因此当我访问亚马逊时，起始页面跟我打招呼时通常只说“你好”。但如果我为了用购物车添加商品而启用 cookie，之后我再访问时它就会说“你好，布莱恩”。有人觉得这种表面上人性化的招呼很讨喜，但对我而言，这种做法时时提醒我完事后要删除 cookie 并重新将其禁用，以限制网站对我的跟踪。

每个 cookie 都有名字，一台服务器可以为一次访问存储多个 cookie。cookie 有失效时间，过期了就会被浏览器删除。浏览器只能将 cookie 发送给当初返回它的域，但是否接受或返回 cookie 并没有强制规定。需要指出的是，cookie 只是一串存储在浏览器中供以后返回服务器的字符，它是完全被动的，不会被返回给来源服务器之外的其他域。cookie 不是程序，也不包含动态内容。

在你的计算机上可以很容易看到 cookie，浏览器会告诉你它管理的所有 cookie，你通常也能在文件系统里找到它们。例如，当我访问《纽约时报》网站时，即使没有登录，

我的浏览器也会保存 15 个 cookie，其中有个叫 WT_FPC 的 cookie 包含了如下能看出含义的信息：

```
id=140.247.62.34-237799664.30112241:lv=1288631855059:ss=1288631855059
```

这里 id 的值像是 IP 地址和标识号，另外两个值怎么看都像是时间，也许是最近一次访问的时间和会话开始的时间吧。

理论上，cookie 的功能都是挺善意的，创造这一技术的本意也确实如此。但总会有些有违初衷的坏事发生，cookie 也被用到了人们不太喜欢的用途上。最常见的就是在用户浏览时跟踪其行为，生成用户访问网站的记录，然后令人生厌地向用户投放定向广告。在下一章，我们将详细讨论这种行为的工作原理以及跟踪用户上网过程的其他技术。

10.5　动态网页

最初设计万维网的时候并没有考虑在客户端运行程序。第一个浏览器可以帮助用户生成请求、发送表单里的信息，并能在辅助程序的帮助下显示图片、声音等需要特殊处理的内容。不久之后，浏览器就能运行从网上下载的、有时被称为**动态内容**的代码了。至于如何执行，要取决于代码本身，跟浏览器也有关系，但基本思路就是把用某种语言写的代码下载到客户端计算机来运行。不难看出，这会带来某些影响，有些是有益的，也有些显然不是。

在第 6 章我们讲过，浏览器就像一个专用操作系统，可以通过扩展功能来处理纷繁复杂的内容，以此“增强你的浏览体验”。但是，为达成上述目的，需要你的浏览器运行别人写的程序，而你对这些程序的“品行”却一无所知。

在浏览器里运行程序的好处是可以增强浏览器的功能，而且当计算在本地完成时，交互会快很多。不利的一面在于，在你的计算机上运行未知来源的代码真的存在风险。显然，“我的身家性命全靠陌生人的良心”并不是靠谱的安全策略。微软有篇文章叫“关于安全的 10 个不变法则”（Ten Immutable Laws Of Security），第一条说的就是：如果有坏蛋能说服你在自己的计算机上运行他的程序，那你的计算机就不是你的了。

Netscape Navigator 的早期版本可以运行 Java 程序。那时候 Java 还是相对较新的语言，被设计成能轻松安装到计算能力不是很强的环境中（比如家电），所以在浏览器里包含 Java 解释器在技术上并没什么困难。这使得在浏览器里进行重大计算工作的前途一片光明：浏览器可能代替文字处理和电子表格这样的传统程序，甚至操作系统本身也可以被浏览器替代。这种前景让微软坐立不安，于是接连出手排挤 Java 的扩张。为此，Java 的创始者太阳微系统公司（Sun Microsystems）与微软多次对簿公堂。

由于各种原因，Java 最终没有成为扩展浏览器的主要途径。Java 本身是一门功能丰富的语言，但与浏览器集成时受限较多，因此现在已经很少将 Java 用于扩展浏览器了[①]。

1995 年，Netscape 还推出了一种专用于其浏览器的新语言 JavaScript。别看名字里有 Java，其实 JavaScript 跟 Java 没有任何关系，唯一能扯上关系的是两者写出来的程序都长得像 C 语言。选择这个名字只是出于市场宣传的目的。[②] Java 和 JavaScript 的实现都使用了虚拟机，但两者的技术差别很明显。Java 源代码在其创建之处编译，生成的目标代码发送到浏览器解释运行，因此你看不到初始的 Java 源代码是什么样子的；而 JavaScript 发送到浏览器的就是**源代码**，整个编译过程都在浏览器里进行，从而使接收到 JavaScript 程序的人能看到要执行的代码。由于接收到 JavaScript 源代码的人不但可以运行它，而且也能研究和改写它，因此，没有办法保护 JavaScript 源代码。

如今的大多数网页都包含一些 JavaScript 代码，用来展示图形特效、验证表单信息、弹出有用的和讨厌的窗口等等。尽管一方面弹窗拦截器减轻了 JavaScript 弹出广告给人们带来的烦恼，以致连浏览器现在都集成了拦截功能，但另一方面，用 JavaScript 进行的复杂跟踪和监控活动也增长迅猛。虽然像 NoScript 和 Ghostery 这样的浏览器附加程序可以控制 JavaScript 代码的可运行功能，但由于 JavaScript 用得太广，如果不允许或限制浏览器运行 JavaScript，我们上网时就会困难重重。尽管有段时间我很担心

---

① 有个很有趣的例子，兴业银行（http://www.cib.com.cn）的“个人网上银行”登录页面里，有脚本检测浏览器类型，如果是 Internet Explorer，就开启一个用 ActiveX（本节稍后将会讲到）写的密码保护控件；对于不支持 ActiveX 的浏览器，则使用 Java 版本。——译者注

② 事实上，JavaScript 的语言特性主要来自 Self 和 Scheme 语言，只是使用了类似 C 语言风格的记法。JavaScript 一开始也不叫这名字，而是叫 LiveScript，后来因 Netscape 和 Sun 合作，才把名字改为 JavaScript 以便市场推广。——译者注

JavaScript 的负面影响，尤其担心网站会用它来跟踪我上网浏览的行为（下一章将会谈到如何跟踪），但权衡下来，我觉得 JavaScript 还是利大于弊。我隔三差五地彻底停用 JavaScript，然后又因为我关注的很多站点用到了这玩意而重新启用。

借助浏览器自己的代码或 Apple QuickTime、Adobe Flash 这样的**插件**，浏览器也可以处理其他语言和内容。插件是按照需求动态加载到浏览器的程序，一般由第三方开发。如果你访问的页面里有浏览器自己不能处理的内容，浏览器会提示你"获取插件"。意思就是要你下载一个新程序，在你的计算机里配合浏览器一起运行。插件能做什么呢？理论上它可以为所欲为。所以你不得不信任插件的发行者，否则就没法显示那些内容。

插件是编译好的代码，通过调用浏览器提供的 API，作为浏览器的一部分运行。常用的插件包括遍地都是的 Flash 动画播放器和微软的 Silverlight 等，后者是取代 Flash 处理视频和其他服务的另一种选择。对于插件，如果我们长话短说，那就是：如果你信任插件的来源，那在使用它的时候，就跟使用其他有瑕疵和会监控你行为的代码没什么两样。HTML 的新版本 HTML5 为浏览器带来了新功能，可以减少对插件的依赖，尤其是在视频和图形方面。但在很长一段时间内，插件还会继续扮演重要角色。

可控性最差或者说最不可控的浏览器扩展技术是微软的 ActiveX，这种技术能让 Internet Explorer 加载代码并运行。ActiveX 像是搅起了江湖血雨腥风的武林秘籍：代码本身不受限制，可以运行任何 Windows 功能，所以实际上能完全控制你的计算机。显然，这是一把双刃剑：用 ActiveX 组件可以实现任何功能，但出错的风险也大大增加——如果代码来自一个坏蛋，造成真正损害的可能性就更大了。由于你没办法知道 ActiveX 组件到底会做什么，于是只好无奈地信任它的发行者。

为此，微软搞出了一种叫 Authenticode 的数字签名机制，想用这种办法来确保代码至少是来自它所声称的发行者而不是别人假冒的。在 Internet Explorer 的安全模式里，也确实有只允许下载和运行已签名的 ActiveX 代码的选项。当然，这种方法还是不能保证代码只执行正确操作，甚至也不能保证代码来源友善，但总比摸黑运行未知来源的代码要好一些。

10.6　网页之外的动态内容

除了网页，动态内容也可出现在万维网的其他地方。随着万维网服务的增长（下一章将会提到），这些动态内容中潜在的问题也愈发严重。试想一下电子邮件吧。邮件到达的时候，会显示在邮件阅读程序里。显然，邮件阅读器一定要显示文本内容，而值得关注的是，如果邮件包含了其他内容，应该解析到什么程度为好，因为这是关系到用户隐私和安全的大事。

邮件正文里有 HTML 会怎样？虽然邮件里出现大号红头文字会让收信人不悦，但显示这样的邮件其实并无危害。邮件阅读器应该自动显示图像吗？这样便于收件人查看照片，但也为更多的 cookie 打开了方便之门。我们可以通过立法来禁止这些行为，但又能用什么办法来阻止发件人在邮件里嵌入 1×1 的透明像素并在其 URL 里编入收件人的某些信息呢（这些看不到的图像有时称为**网页信标**，web beacons[①]）？支持 HTML 的邮件阅读器会按 URL 请求这些图像，从而使存放图像的网站得知你在某个时候读了这封邮件。通过跟踪你阅读邮件的时间，有可能获得你本想保密的信息。

如果邮件阅读器能处理 HTML，那它也要解析 JavaScript 吗？最近我试用了一个新推出的基于网页的邮件阅读器。令人惊讶的是，它不但解析了邮件里的 JavaScript，甚至按其默认设置在不提示我的情况下，就自作主张地执行了这些代码。

如果邮件里包含 Word、Excel、PowerPoint、PDF 文档或 Flash 电影，又会发生什么呢？邮件阅读器应该自动运行这些程序吗？还是退而求其次，但为了方便你使用，仍然允许在邮件某个位置点一下就运行呢？再多想一步，应该让你直接点击邮件里的链接吗？要知道，这可是引诱你上当的好把戏。此外，PDF 文档也可以包含 JavaScript（第一次听说时，我也吃了一惊），如果邮件阅读器自动调用 PDF 阅读器，那后者又是否应该自动执行 PDF 文件里的 JavaScript 代码呢？

在邮件里附加文档、电子表格、幻灯片是非常便利的，这也是商业社会的标准做法。但考虑到这些文档可能携带病毒，我们马上会看到，不分青红皂白地打开附件的做法会助长病毒的蔓延。

---

① “网页信标”是中性说法，也有贬称为“网页臭虫”（web bugs）的。——译者注

还有更糟糕的状况——邮件里包含可执行文件，比如 Windows 的.exe 文件之类的。点击这样的附件就会启动这些程序，极有可能在运行后给你或你的系统带来危害。网上的坏蛋们会用各种伎俩骗你运行这些程序。我曾经收到一封邮件，声称里头有俄罗斯网球运动员安娜・库尔尼科娃的照片，鼓励我打开看看。该文件的名字是 kournikova.jpg.vbs，但是扩展名.vbs 隐藏了起来（Windows 默认隐藏扩展名，这个设计害人不浅），收件人很难发现它并非照片而是 Visual Basic 程序。幸好我用的是 Unix 系统下的老古董邮件程序 Pine，只支持文本模式，无法直接点击运行。于是我把“照片”另存为文件待稍后检查，这个程序也就此露出了马脚。

10.7　病毒和蠕虫

上一节提到的安娜・库尔尼科娃“照片”实际上是个病毒。下面我们就谈谈病毒和蠕虫。这两个词都指在系统间传播的、通常是恶意的代码。两者在技术上有个细微的差别：病毒的传播需要人工介入，也就是只有你的操作才能催生它的传播；而蠕虫的传播却不需要你的援手。

虽然出现这种程序的可能性早就为人所知，但第一个在晚间新闻节目中亮相的则是罗伯特・莫里斯（Robert T. Morris）写的“互联网蠕虫”。该程序在 1988 年 11 月被放到网络上，而那个年代还不是现代意义上的互联网时代。莫里斯的蠕虫用两种方法把自己从一个系统复制到另一个系统：一是利用常用程序的缺陷，二是用字典攻击（即用常见词来猜测别人可能使用的密码）自动登录系统。莫里斯此举并无恶意，他当时只是康奈尔大学的一名研究生[①]，想设计一个程序来测量互联网的规模，但一个编程错误让蠕虫的传播速度超过了他的预期[②]，结果很多机器遭遇多次感染，无法处理暴增的流量，只好从互联网断开。当时，莫里斯被指控违犯了制订不久的《计算机欺诈与

① 为了不暴露身份，他当时是用麻省理工的计算机来做这事的。有意思的是，1999 年麻省理工聘任他为副教授，并在 2006 年授予他终身职位。此外，他还是 Viaweb 和著名风投公司 Y Combinator 的创始人之一。——译者注

② 蠕虫中有一段代码用来检测计算机是否受到感染，如果没有才继续运行并传播之。但这样很容易通过伪造传染检测标志来免疫，于是莫里斯修改了代码，使得即便已感染仍有 1/7 的概率进行传播，由于扩散趋势是指数级的，这种传播概率还是太高。

——译者注

滥用法案》，被处罚金上万美元并参加 400 小时的社区服务。

在互联网广泛使用之前，软盘是在 PC 之间交换程序和数据的标准介质。病毒因此通过受感染的软盘，肆虐传播了多年。当计算机加载受感染的软盘时，隐藏在其中的病毒自动运行，把自己复制到本地计算机，并进一步感染后续插入的新软盘。

随着微软在 Office 程序尤其是 Word 中包含 Visual Basic，病毒的传播更加容易了。由于 Word 中内建 VB 解释器（至今仍在），Word（.doc、.docx 文件）、Excel 和 PowerPoint 文档中可以包含 VB 程序，因此写个程序在文档打开时获取控制易如反掌。而且，由于 VB 能访问 Windows 操作系统的全部功能，这类程序可以在操作系统中恣意妄为，做任何事。一般来说，病毒会首先安装在本机上，然后想方设法传播到别的系统上。一种常见的病毒传播模式是：病毒把自己附在一封写着无害或诱惑言辞的电子邮件里，寄给被攻击邮件地址簿里的每个人。安娜·库尔尼科娃病毒就是这么做的。如果收件人打开附件里的文档，病毒就把自己装进新系统，进而传播开来。

这样的 VB 病毒在 1990 年代中后期出现过很多。由于 Word 的默认策略是不经用户允许就自作主张地运行 VB 程序，这类病毒传播得非常快，致使很多大公司必须关掉所有计算机、逐台查杀才能消灭病毒。VB 病毒尽管现在还有，但只需改变 Word 和其他同类程序的默认行为就能严重削弱它们的破坏力。另外，现在的很多邮件系统收到新邮件时，会先剔除其中的 VB 程序和其他可执行内容，然后才把邮件送达收件人。

VB 病毒很容易编写，以致写这些东西的人被称为“脚本小屁孩”。实际上，编写一个“作案”时不会被抓到的病毒或蠕虫是很难的。2010 年底，在伊朗的一些过程控制计算机中发现了一个叫“震网”（Stuxnet，也称为“超级工厂”）的复杂蠕虫。它的主要攻击目标是伊朗的铀浓缩设备。“震网”用了一个很狡猾的攻击方法，即控制离心机的转速波动，看上去只会引起正常磨损，却会导致离心机损坏甚至报废；同时，对监控系统报告说运行正常，于是没人注意到离心机出现问题。至今，仍然没有人站出来承认对此负责 。

特洛伊木马（在网络安全的语境中，通常简称木马）是伪装成有益或无害，但实际上有害的程序。因为看起来有一定用处，所以受害者会禁不住诱惑而下载、安装木马。常见的情况是，木马程序自称要对用户的系统进行安全分析，实则是安装恶意软件。

前面提到过，软盘曾是病毒的传播介质。随着存储设备的更新，现在这一角色开始由 USB 闪存驱动器接替。也许有人认为闪存驱动器只有存储功能，是个被动设备，很难传播病毒。然而，以 Windows 为代表的一些系统，都有个“自动运行”的功能：当插入 CD、DVD 或闪存驱动器时，系统会自动从盘上运行某个程序。恶意程序会在没有警告的情况下自动安装，造成令人措手不及的破坏。即便大多数公司规定严格的安全策略，限制在公司计算机插入 USB 驱动器，但还是有相当多的系统通过这种方式遭受感染。个别情况下，甚至新买的驱动器里都会带有病毒。

10.8 万维网安全

万维网引发了很多安全难题。一般来说，可以把万维网遇到的安全威胁分成三类：对客户端（也就是使用浏览器的你）的攻击、对服务器（例如网上商店或者银行帐户）的攻击和对传输中信息的攻击（比如窃听无线网络）。我们将在本节依次讨论这三种情况，以弄清问题出在哪里、如何减轻危害。

10.8.1 对客户端的攻击

对你的攻击不仅包括垃圾邮件、跟踪等惹人讨厌的攻击，还包括泄露你的信用卡和银行账号等私密信息、假冒你花掉你的钱等更严重的攻击行为。

下一章我们会详细讨论 cookie 和其他跟踪机制如何监控你的上网行为、给你发送看似吸引人从而不是那么烦人的广告。实际上，通过禁用第三方 cookie（也就是从别的网站而不是你正在访问的网站发送过来的 cookie），就可以堵住大部分跟踪攻击。如果再使用其他浏览器插件来禁止跟踪程序运行、关闭 JavaScript、禁用 Flash，采取这些防护手段就能把跟踪和广告的数量限制到可控的程度。也许有时你会嫌这些防护手段麻烦，因为这样披挂上阵会导致有的网站无法访问。遇到这种情况时，就要把防护的级别临时调低一些，但要记得访问完这些网站后再复原。

垃圾邮件是不请自来的邮件，其内容通常是发家致富秘籍、洗髓易筋偏方、职场蹿升宝典和很多没用的其他商品和服务。垃圾邮件业已泛滥成灾，严重影响了电子邮件的正常使用。我自己通常每天收到 50 到 100 封垃圾邮件，远比真正的邮件多得多。据

可信调查称，所有电子邮件中的 90%是垃圾邮件。最近我看到一篇文章则说，万维网上每天发送的垃圾邮件多达 2500 亿封。垃圾邮件之所以如此泛滥，主要原因在于发送成本极低（上面那篇文章里说，发送 100 万封垃圾邮件只要 80 美元成本）。就算几百万收件人里回复的只占个零头，也够那些发件人赚回来了。

可以使用垃圾邮件过滤器对邮件进行分拣，去芜存菁。常用过滤方法有查找已知的垃圾邮件模式（比如“你想每天都收到 eBay 寄来的钱吗？”这样的邮件，一看就是群发给大量人群的格式邮件），发现不太可能出现的名字和离奇的拼写（比如\/I/-\GR/-\）[①]，或者把经常发送垃圾邮件的地址列入黑名单，拒收它的邮件。只用一种办法来过滤是远远不够的，因此要打出组合拳。垃圾邮件过滤仿佛是军备竞赛，只要防守者道高一尺，学会防范某一类垃圾邮件，进攻者就会魔高一丈，发明出新的规避方法。

要想从源头制止垃圾邮件非常困难，因为垃圾邮件的源头通常都极其隐蔽。很多垃圾邮件是通过被入侵的个人计算机（通常运行 Windows 系统）发送出来的。由于存在安全漏洞，再加上用户疏于管理，这些计算机很容易被装上**恶意软件**（malware）。有一种恶意软件会在收到上游控制指令时，群发垃圾邮件，而它的上游计算机也可能进一步受更上游控制。每多一步都会使发现垃圾邮件的源头更加困难。

网络钓鱼（phishing）攻击者通过骗取收件人的信任，让他们主动奉上窃贼需要的信息。你十有八九收到过“尼日利亚骗局”式的诈骗邮件。真的会有人相信这些天方夜谭，给骗子回信吗？听起来不可思议，但确实一直有人上当。钓鱼攻击比这更狡猾，因为它们发来的邮件看起来就像来自正规金融机构甚至你的朋友们。试想，如果你收到一封道貌岸然的邮件（像是你熟知的公司发来的）让你核实自己的证件……如果你照着做了，坏蛋就会知道你的某些信息，以此来窃取你的钱财或者套取你的真实身份。几年前，这可是相当前卫的玩法。我曾收到这样的一封钓鱼信，行文措词很像那么回事儿，可是我的名字不叫比尔，我也没有花旗银行账号。

> 发件人: 花旗银行 cash@citibank.com
>
> 收件人: bill@research. bell-labs.com
>
> 主题: 请更新您的花旗账号

---

① “\/I/-\GR/-\”是用象形字符拼写的一种药物名 Viagra（Viagra 在中国大陆正式译名“万艾可”，更广为人知的民间译名是“伟哥”），这么拼写是为防止关键词过滤。——译者注

亲爱的花旗客户：

我行最近发现有人从国外IP地址频繁尝试登陆您的花旗银行账号,我们一定能相信这是在试图用暴力破角您的密码。登陆成功没有，您的账户已受到全面保护。如果您最近在旅行中访问了自己的账户，这些异常的登陆也许是您自己。

发起登陆的IP地址是173.27.187.24,ISP主机是cache-82. proxyserver.cis.com
我们使用了多种技术来核兑用户在我们网站注册时填写信息的准确性。然而，油于通过互联网核实用户有困难，花旗银行无法做到也并不会确认每位用户所声称的身份。因此，我们起用了一套离线验证系统帮您评估与您交易的是谁。这套系统叫做旗安宝，是花旗银行迄今为止最安全的钱包。如果您是此账户的合法人，只要点击下面的连接，填写表单并提交，我们就会核实您的身份并为您免费注册旗安宝系统。这样，您在我行的所有账户都会受到保护，远离遭受欺诈的风险。

只要轻轻一点，远离欺诈风险!

为了让Citibank.com成为最安全的网站，每一位用户都要被注册到旗安宝。
注意！如果您不理会我们的请求，我们在别无选择之下将不得不暂停您的账号。

另，请勿回复此邮件，这个账号不能收邮件。

此致，敬礼。花旗银行客户支持部。

要编造一封看起来正规的邮件是很容易的，版式和图片都可以从真的网站上复制。使用无效的回信地址也很容易，因为邮件系统发件时并不检查发件人的真实性。那个链接下面的 IP 地址指向一个美国之外的网站，而我是在检查了 HTML 源代码后用`traceroute`才发现这一点的。和垃圾邮件一样，钓鱼也几乎不花任何成本，即便上钩的人少之又少，也足够骗子们赚个盆满钵溢了。

更新式的攻击则更难发现。2010年末，我收到了一位朋友发来的这封信，该信是群发给一个很小的私密邮件列表的：

日期: Wed, 1 Dec 2010 04:55:37 -0800 (PST)

发件人: 查尔斯 [...] <[...]@gmail.com>

实在不好意思凿紫打扰你，但是我刚遭遇了人在囧途之英国囧，包包被偷，护照和银行卡随之不见。幸好大使馆为人民服务，给我搞了张临时护照。我现在得买张票，还要把旅馆账单结了。

就不跟你客套了，现在手头没钱，跟银行联系过，他们说寄给我一张新卡，最快也要 2 到 4 天才能到手，还得是万恶的工作日！我看还是跟哥们儿你借点闲钱吧，回来马上还你，我得去赶几个小时之后的末班飞机。

一会儿告诉你怎么打钱给我。你可以发我的 AOL 账号([...]@aol.com)，我拿手机登在上面刷着，要不打旅馆的座机也行，号码是+447045749898。
万分感谢！

查理

看上去真是像模像样啊。查理是个驴友，在外旅行是常态，我已经有好几个星期没见过他了。于是我满怀疑虑地打了那个电话，没人接，有点奇怪。然后我回了那封信，收到他的回复：

嗨，布莱恩。

我有点小麻烦，借我 1800 美元吧。用西联打钱就行，需要汇款操作细节的话你就说。回国以后马上还你。

查理

我问他要了汇款指南，同时也问了个问题，只有查理本人才能答出来。他回给我西联的详细操作步骤，却没搭理我的那个问题。我顿时疑心大起，马上搜索了一下那个电话号码，发现是一起知名的骗局，已经连骗了好几个月。骗子竟然连剧情和电话号码都懒得换。后来我得知，查理的 Gmail 帐号遭遇过入侵，攻击者把他的联系人名单作为下手目标，并了解到他的一些私人信息。因此，这种精确瞄准的攻击方式有时叫做**鱼叉式网络钓鱼**（spear phishing）。鱼叉式网络钓鱼是一种**社会工程**方法，即假装是受害人的私交密友，或者冒称同事，诱使受害人犯傻。

间谍软件（spyware）是指运行在你的计算机上、会把你的信息送到别处去的程序。有些间谍软件显然是恶意的，但有些只是为了商业目的而私自收集用户信息。例如，我有几台笔记本预装了包探测器程序 pinger，该程序开机自动运行，然后定期连接到厂家的服务器。至少在我某天发现并将之禁用之前，它一直在这么做。我相信如果质问厂家的话，他们一定会告诉我这个包探测器只是向他们报告我机子持续运行时的健康状态并检查软件更新。问题是：我从没收到关于这个程序的说明，也没见过打开这个功能的选项，更不要说给我机会关掉它了。

还有一个臭名远扬的例子。索尼公司在 2005 年卖出过一批音乐 CD，其中使用了自动运行技术，能在 Windows 系统上安装间谍软件：光盘插入计算机后，就会悄无声息地自动安装监控程序。这个间谍软件用了一些手法来隐藏自己，这说明某些人也知道这样做不够厚道。这个程序的实现有缺陷，会让计算机完全开放，导致任何人都能安装软件。索尼这样做想必并非出于恶意，而只是为了防止非法复制。但这个程序发现计算机播放音乐时，不管是不是索尼发行的，都一股脑报告给索尼。这个做法曝光之后，索尼的反应是发布了一个让安全性变得更加糟糕的卸载程序。

此外，入侵者往往会在个人计算机上安装**僵尸**（zombie）程序，这类程序平时潜伏着，一旦控制者从互联网发来指令，就会发作并执行诸如发送垃圾邮件等攻击。这些程序一般也叫**肉鸡**（bot），控制它们的各个网络通常叫**僵尸网络**（botnet）。万维网上已知的僵尸网络有上千个，可能控制了几十万到上亿台的肉鸡待命出击。

攻击者入侵客户端计算机之后，就可以查看其文件系统，或者悄悄安装**按键记录器**抓取用户输入的口令和其他数据，这样就能从源头窃取信息。按键记录器是在客户机上监控所有按键行为的程序，所以能抓到用户输入的口令。而且在这种情况下，加密也没用[①]。这种程序也可能开启计算机的话筒和摄像头。

如果浏览器或别的软件有缺陷，导致不怀好意的人在你的机器上安装了他们的软件，风险就更大了。浏览器程序庞大而复杂，存在种种可能导致用户遭受攻击的缺陷。为此，要让浏览器始终更新至最新版本，还要配置浏览器让它不要向外发送不必要的信

① 有些网站为了防止按键记录器窃取用户敏感信息，自己开发了 ActiveX 控件处理口令输入，但这样做有严重的负面影响。参见：http://t.cn/h5VPkQ。——译者注

息，也不要允许随意下载。另外，请注意你自己下载回来的东西，不要仅仅因为网页或程序让你点鼠标，就盲目去点。稍后我们将讨论更多的防御措施。

10.8.2　对服务器的攻击

对服务器的攻击并不是用户自己的问题，因为普通用户对此无能为力。但覆巢之下无完卵，服务器遇到这类攻击时，普通用户也将沦为受害者。

为阻止对服务器的攻击，服务器的编程和配置必须非常仔细，不论客户端发来的请求构造得多么精心，都要确保不泄露未授权信息，不允许未授权访问。但实际上，服务器上运行着的大型复杂程序，使服务器容易出现可能被攻击者利用的编程缺陷和配置错误。**SQL 注入**（SQL injection）就是一种常见的针对服务器的攻击方式。在万维网上，服务器一般在后台运行数据库，访问数据库时通常使用标准接口 SQL（Structured Query Language，结构化查询语言）。如果没有严格限定访问权限，机灵的攻击者就可以提交精心构造的 SQL 查询指令来检查数据库结构，从而提取未授权的信息，甚至有可能在服务器上运行他们自己写的代码，进而控制整个系统。尽管上述攻击及其防御机制是众所周知的，但服务器被攻击的事件仍然不断发生。

系统一旦沦陷，能制约入侵者恶行的限制就很少了。特别是当入侵者设法获得了 root 权限（也就是系统的最高级别管理权限）时，他们就更会为所欲为。不论是对服务器，还是对你家里的个人计算机，都是如此。获得服务器 root 权限后，入侵者就能摧毁网站，或者在网站上发布令人尴尬的言论以丑化所有者的形象。还可能下载破坏性程序，或者在网站上存储并发布色情图片和盗版软件等违法内容。服务器可能会因此丢失大批数据，个人计算机一样在劫难逃。2011 年 4 月，黑客从索尼 PlayStation 网窃取了七千万用户的姓名、家庭住址、电子邮件地址等信息。在几乎同一时期的另一桩案子中，为多家大企业提供电子邮件营销服务的 Epsilon 公司丢失了几百万用户的姓名和电子邮件地址。（两家有业务往来的公司都就 Epsilon 的事故给我发送了提醒，一家说只泄露了电子邮件地址，另一家则说还泄露了姓名。两家说法不一致让我谁的说法也没法相信了。）类似这种事故，虽然规模未必总是这么大，却经常发生。

服务器还经常遭遇**拒绝服务攻击**（DoS，Denial of Service）。在这种攻击中，发起者把大量网络流量引导到一个网站，利用密集的访问使其停止响应。拒绝服务攻击是精心

策划的，通常用僵尸网络来完成：沦陷的肉鸡接到命令后，会在指定时间访问指定网站，从而导致流量骤增。从多个来源同时发起的拒绝服务攻击通常称为**分布式拒绝服务攻击**（DDoS，Distributed Denial of Service）。

不幸的是，防御者需要防住所有可能的攻击，而攻击者只要找到木桶的一块短板就行。因此，这场攻防战的双方是不对等的，天平倾向于攻击一方。

10.8.3　对传输中信息的攻击

尽管**对传输中信息的攻击**仍然很严重也很常见，但这个问题在万维网安全中往往是放到最后考虑的。无线网络的普及，或许会提高人们对这个问题的重视程度。例如，坏蛋可能会偷听你和银行的对话，窃走你的帐号、口令等信息。但如果你和银行之间的通信是加过密的，偷听者就很难分析出他听到的数据是什么意思了。HTTPS 协议是加强型的 HTTP，对请求和应答双向数据都做了加密。这样，偷听者通常就不能读取信息，也无法伪装成通信中的任何一方。然而，由于 HTTPS 并没有被广泛使用，攻击者仍然可以用 Firesheep 之类的程序在咖啡馆、机场和提供无线上网服务的其他公共场所偷听连接，从而可以让他们假扮成你，而你根本无法察觉到。靠窃听商店里未加密的终端机通信就可以获得信息卡信息：窃贼坐在商店外的车里，一有交易发生，窃听设备就可以捕获信用卡数据。

还有一种攻击形式是“中间人攻击”，即攻击者截获消息，并按自身需要加以修改后发出去，该消息看起来是直接来自源头的。（第 8 章讲到的《基督山伯爵》中的一个故事，采用的就是这种攻击手法。）

VPN（Virtual Private Network，虚拟专用网）在两台计算机间建立起加密的通道，以保证其间双向通信的安全[①]。企业通常通过部署 VPN 来让员工在家里办公，或者到国外出差时连回公司。用户登录 VPN 不仅需要输入口令，还需要持有一个物理设备。这种“双因子”认证比只输入口令要安全，因为它不仅需要用户知道某些信息（口令），还需要拥有某些实物（设备）。在实际应用中，这个设备可以是手机里运行的程序，

① 第 9 章曾经提到，网关也是一种专用的计算机。如果两个网络不直接相连，也可以用 VPN 在它们各自的网关之间建立加密通道，从而实现网络到网络的安全通信。——译者注

能按某种算法生成动态数码以便跟服务器上用同样算法生成的数码匹配，也可以是一个能显示数码的专用设备，在输口令时把设备显示的数码一起输进去。双因子认证技术不仅用于 VPN，也可用于银行和其他在线账号服务。



颇具讽刺意味的是，制造出业界广泛使用的双因子认证设备 SecureID 的 RSA 公司[①]在 2011 年 3 月遭遇入侵，安全信息失窃，从而使某些 SecureID 设备不再安全。随后，军火商洛克希德·马丁公司受到攻击，看样子黑客充分利用了 SecureID 失窃案中得到的信息。

10.8.4　自我防御

个人计算机用户如何进行自我防御？有人向我征求这个意见时，我会用本节的内容告诉他。我把计算机用户的防御手段分成三类：第一类非常重要，第二类中等重要，第三类则要看你的偏执程度。（如你所料，第三类我就不细讲了，因为大多数人根本不会偏执到这种程度。）

非常重要

选择安全的口令，让别人不会轻易猜中，即使反复尝试也不至于马上就被破解。一串随机的大小写字母、数字、特殊字符混搭而成的口令，显然要比词典里的单词、生日、家人和宠物名来得安全。至于信奉“最危险的才是最安全的”，从而用“password”作为口令的搞笑念头，还是及早打住为妙。

不要在重要网站和无关紧要的网站用同一口令，前者包括银行网站和电子邮件，后者譬如在线报纸。工作账号和私人账号不要用同一口令。经常修改口令，但不要改得太

① 即 10.9.2 节提到的 RSA 加密算法的三个发明人所创建的公司，现已被 EMC 收购。——译者注

有规律，比如在最后一位数加 1。

不要用开放的无线网络做重要的事。保证用 HTTPS 连接无线网络，但别忘了 HTTPS 只加密内容。

收到来历不明的邮件不要打开附件。收到朋友寄来的邮件，如果发现有意想不到的附件，也不要打开。不要因为软件提示你接受、点击或安装就全部照做。不要从可疑来源下载软件，除了可信来源之外，下载和安装软件都要保持警惕。

使用 Windows 要安装防毒软件并保持更新，不要去点那些声称要对你的计算机进行安全检查的网站链接。关闭微软 Office 程序的宏，尽量禁用 ActiveX 控件。如果用的是苹果电脑，现在还不太需要防病毒软件，但同样要小心。其他软件，比如浏览器和操作系统本身，既然厂家经常发布安全补丁，就可经常更新。

中等重要

关闭浏览器弹窗和第三方 cookie（讨厌的是，每个浏览器都分别保存自己的 cookie，所以每个浏览器要分别设置一遍）。用 Adblock 或 Flashblock 等软件拒绝广告图片。

用垃圾邮件过滤器筛选电子邮件。

关闭 Adobe Reader 的 JavaScript。

关掉不用的服务。比如我的苹果电脑可以共享打印机、文件和设备，还能从别的计算机远程登录进来管理我的计算机。Windows 也有类似的一套服务。我只保留了打印机共享，其他统统关掉了[①]。

打开计算机上的**防火墙**程序，监控进出的网络连接，禁止违反访问规则的连接。

只要有可能，就对重要账户使用双因子认证。

---

① 需要注意的是，网上流传的一些关闭 Windows 服务的“教程”（尤其是转来转去不知出处的中文教程）不求甚解、以讹传讹，而且通常是为了“精简”系统运行时的资源占用，不是出于安全考虑，甚至还为了躲避正版验证而关掉自动更新，因此使用 Windows 的服务管理器关闭服务时务必慎重。——译者注

偏执狂专用

禁用邮件阅读程序读取 HTML 和 JavaScript。

用 NoScript 限制 JavaScript，用 Ghostery 限制跟踪。

关掉所有 cookie，只对自己明确想启用的网站打开。

选用那些不易成为攻击目标的系统，比如用 Linux 或 Mac OS X 系统而不是 Windows。使用 Chrome、Firefox、Safari 和 Opera 等浏览器，而不使用 Internet Explorer。

鉴于手机也越来越多地成为攻击目标，在手机上采用类似的预防措施也是很必要的。

10.9　密码术

密码术被称为“秘密书写”的技艺，用以和别人交换私密信息，已有数千年历史。尤利乌斯・凯撒大帝曾用过一种简单的密码方案（后人称之为凯撒密码）：把消息原文中的字母向后移动三个位置，于是 A 变成 D、B 变成 E……，“HI JULIUS”就变成了“KL MXOLXV”[①]。这种算法现在仍然有生命力，有个叫 `rot13` 的程序就是把字母移动 13 个位置进行变换，常用在新闻组里，隐藏冒昧言论或者剧透，以防被人意外看到，并不是真的用来加密。

在人类运用密码术的历史长河中，涌现出了丰富多彩故事。有些轻信加密就能保密的主人公，还曾因此丢了性命。1587 年，苏格兰女王玛丽就因为加密不当而命丧断头台。当时，她和密谋罢黜英格兰女王伊丽莎白一世、把她本人推上英格兰王位的同谋者通信，然而密码系统被破解，阴谋内容和同谋者名字在中间人攻击下暴露出来，他们在劫难逃，被斩首示众。1943 年，由于日本的军用加密系统不够安全，日本联合舰队总司令山本五十六大将一命呜呼。美国情报机关破解了山本的飞行计划，于是派飞行员击落了他的座机。夸张一点说，正是由于英国人靠阿兰・图灵的计算知识和技术破解了恩尼格玛密码机，解密了德军的军事情报，才使得二战结束时间大大提前。

① 这里作者用现代英语字母举例，事实上古拉丁语字母并没有 J 和 U。——译者注

加密的基本思路是，张三和李四[1]想互相交换消息，要求通信内容保密，但并不掩盖他们正在通信的事实。要做到这一点，他们需要共享用来扰乱以及随后恢复要传递消息的同一段密文，这样，别人就看不懂他们写了什么，只有两位当事人自己看得懂。用来加密消息的密文被称为**密钥**。例如，在凯撒密码里，密钥就是字母移位的距离，3 表示把 A 换成 D。对于恩尼格玛密码机这种复杂的机械加密设备来说，密钥就是若干代码转子设置和一组插头接线方式的组合。基于计算机的现代密码系统则使用巨大的秘密数字作为密钥，把秘密数字作为变换消息中比特流的复杂算法的输入。这样，如果不知道这个数字，就不可能还原消息。

有多种方法用来对加密算法实施攻击。频率分析法统计密码中每个符号出现的频率，可以轻松干掉凯撒密码和报纸上填字游戏所用的简单替换密码。要抵御频率分析，加密算法必须要做到让密文中所有符号都以大致相等的机会出现，以做到没有模式可供分析。但有时候，攻击者可能知道与待破解密钥加密的密文相对应的明文；即使不知道，他们还可能会选取一段明文，诱使被攻击者用待破解密钥加密这段明文，从而两相对照，达到破解的目的。好的算法要能有效抵御所有这些攻击。

现代密码学必须假定攻击者知道并完全理解密码系统的工作原理，从而将所有的安全性都寄托在密钥上。与此相反的做法是假定对手不知道系统用什么加密方案、如何破解，这被称为**隐匿式安全**，只要时间一长，这种做法肯定要完蛋。苏格兰女王玛丽、山本五十六和德军都没有重视这个至关重要的事实，并为此付出了致命的代价。实际上，如果有人鼓吹他们的加密系统十分安全，却不愿说出其工作原理，那就可以确信它并不安全。

这年头，加密故障通常不会给身体带来危险，但会让那些盲目相信加密系统没有缺陷的人颜面扫地。最好的例子是用来加密电影 DVD 的内容扰乱系统（Content Scrambling System，CSS）。1999 年，一位年仅 15 岁的挪威学生约恩·莱克·约翰森（Jon Lech Johansen）发布了破解 DVD 的 DeCSS 程序，随后这个程序广为流传，他也被人们称为“DVD 约恩”。后来约翰森再接再厉，又用逆向工程破解了一些别的加密系统。

① 原文使用的人物角色是爱丽丝（Alice）和鲍勃（Bob），这是英文资料在描述加密算法时常用的两个虚构角色，其英语名字分别以 A 和 B 开头，表示最先介入通信的两个人，类似汉语中的“张三和李四”。其他角色详见 http://en.wikipedia.org/wiki/Alice_and_Bob 的描述。——译者注

目前使用的加密系统基本可分成两类。一类是历史较长的**密钥加密**，也称对称密钥加密，因为加密和解密要使用相同的钥匙。“对称”这个词能更好地描述其本质，但“密钥”则让人更容易区分它与另一类加密系统：**公钥加密**。

10.9.1 密钥加密

在密钥加密系统中，使用同一个密钥对消息进行加密和解密，这个密钥由参与消息交换的各方共享。假定选用的算法完全为人所理解，而且也没有任何缺陷和弱点，那么破解消息的唯一方法就是**蛮力攻击**（brute force attack），即尝试所有可能的密钥，直到找出用来加密的那个。蛮力攻击很耗时间，如果密钥有 N 位，穷举时间就和 2^N 成正比。然而蛮力攻击也并非没有用，因为攻击者可以先尝试短密钥，再尝试长的；先尝试可能性大的密钥，再尝试可能性小的。比如**字典攻击**（dictionary attack）就是尝试用“password”和“123456”这些常见单词和数字形式攻击。如果我们选择密钥时偷懒或粗心，就会让这种攻击得手。

从 1976 年到 2000 年代初，最常用的密钥加密算法是 DES（Data Encryption Standard，数据加密标准），该算法由 IBM 和 NSA（National Security Agency，美国国家安全局）共同开发。尽管有人怀疑 NSA 在 DES 里埋了一个秘密的陷门机制，这样他们就能轻松破解用 DES 加密的信息，但这种怀疑从未得到证实。DES 总是使用 56 位密钥，随着计算机的运算速度越来越快，这个密钥长度显然已经太短了。早在 1999 年，计算机使用蛮力攻击，穷举一天就能破解 DES 密钥。一些使用更长密钥的新加密算法应运而生。

这些算法中使用最多的是 AES（Advanced Encryption Standard，高级加密标准）。该算法是为解答一道全球公开竞赛的题目而开发的，竞赛的赞助方为美国国家标准技术研究所（National Institute of Standards and Technology，NIST）。当时，来自全世界的参赛选手提交了几十个算法。经过激烈的公开评测，比利时密码学家琼・德门（Joan Daemen）和文森特・赖伊曼（Vincent Rijmen）发明的 Rijndael 算法摘取桂冠，并于 2002 年成为美国政府的官方标准[①]。这个算法已归入公有领域[②]，任何人都可以无偿使

① 严格说来，最终的 AES 标准是 Rijndael 算法的特例，因为 Rijndael 允许的密钥长度比 AES 更灵活。——译者注

② public domain，是一个知识产权概念。对于公有领域内的知识财产，任何个人或团体都不具所有权益，这些知识属于公有文化财产，任何人可以不受限制地使用和加工它们。作品归于公有领域的原因一般包括无相关法律保护、专有权利期满、作者自愿放弃、作者无资格占有等。——译者注

用。AES 支持 128、192 和 256 位三种密钥长度，可能的密钥数量非常多，用蛮力攻击算很多年也不会有结果，除非能发现算法有弱点。

AES 和其他密钥加密系统面临的一个大问题是**密钥分发**：参与秘密通信的每一方必须知道所用的密钥，所以必须有绝对安全的渠道把密钥送到通信参与方。这看似很简单，就像把大家都请到家里来聚餐一样，可如果来的人里有间谍或敌人，显然就无法保证分发密钥时的安全性了。还有一个问题是密钥增生：要保证与彼此间无关的多方相互独立地秘密会话，就要为每组会话准备不同的密钥。这就导致密钥分发更加困难。密钥加密系统的上述困难导致了公钥加密的诞生，我们将在下一节讨论。

10.9.2　公钥加密

公钥加密是怀特菲尔德・迪菲（Whitfield Diffie）和马丁・赫尔曼（Martin Hellman）在 1970 年代中期发明的，采用了与密钥加密完全不同的思想。差不多同时，英国一所情报机构的密码学家詹姆斯・埃利斯（James Ellis）和克利福德・柯克斯（Clifford Cocks）也独立发现了同样的方法，但他们的工作被视为绝密，不能发表，因此与大部分荣誉失之交臂[①]。

在公钥加密系统里，每个人都有一个**密钥对**，包含一个公钥和一个私钥。这对密钥是在数学上有关联的整数，具有如下性质：用其中一个密钥加密过的消息只能用另一个密钥解密，反之亦然。在公钥加密系统里，只要密钥足够长，不论是蛮力攻击还是从一个密钥推算另一个密钥，都将是徒劳的。

在实际使用中，公钥真的是开诚布公：任何人都能拿到，一般是公布在网站上；私钥则一定要金屋藏娇，只有这个密钥对的主人才知道它。

设想以下场景：张三想给李四发一条消息，该消息要加密，保证只有李四自己才能看到。张三先到李四的网站上找到他的公钥，用这个公钥把消息加密后发给他。消息发出的时候，王五在偷听，他发现张三正在给李四发消息，但因为内容加密了，所以不知道发出的是什么内容。

---

① 实际上，埃利斯和柯克斯的成果比他们的美国同行早了 3 年。但由于保密的原因，在 1990 年代末才得以发表。两人为此获得了 IEEE（美国电子与电气工程师协会）的里程碑奖。——译者注

李四能用自己的私钥解开张三发来的消息。私钥只有李四自己知道，而且只有用这个私钥才能解开用他的公钥加密过的消息。反过来，如果李四要给张三发一条加密的回信，就要用**张三的**公钥来加密该消息。这次，王五仍然知道有消息发回来，但他同样只能看到加密过的密文，不知道内容写的是什么。张三收到消息后，用只有自己知道的私钥解就可以解开李四的回信。

由于不需要传送共享的密钥，公钥方案有效解决了密钥分发的问题。张三和李四把各自的公钥放在网站上，任何人都可以给他们发送秘密消息，既不需要事先商定，也不需要交换任何密钥。实际上，参与通信的各方甚至根本不需要认识。

公钥加密是在互联网上进行安全通信的关键要素。设想我上网买一本书。我得告诉亚马逊网站我的信用卡号，但我并不想发送明文，这样就需要使用加密的通信通道。直接使用 AES 是不可能的，因为我和亚马逊之间没有共享密钥。为了商定共享密钥，我的浏览器必须首先随机生成一个临时口令，并用亚马逊的公钥加密这个临时口令，然后把加密后的结果传给亚马逊。亚马逊用它的私钥解出临时口令后，我的浏览器和亚马逊就可以在 AES 算法中用这个临时口令来加密要传送的信息（比如我的信用卡号）。

公钥加密的主要缺点是算法的运算速度慢，比 AES 这种密钥加密算法要慢好几个数量级。所以，通常不会用公钥加密算法加密全部数据，而是分两步走：先用公钥加密协商出一个共享的会话密钥，再用 AES 加密接下来批量传输的数据。

上述加密过程的两个阶段——首先是用公钥加密算法设置临时口令，然后再用 AES 交换大量数据——都是安全的。当我们访问网上商店、电子邮件服务和很多别的网站时，都要跟这种技术打交道。你能在浏览器上看出它在运行，因为浏览器会显示正在用 HTTPS（即安全的 HTTP）协议连接，还会显示门锁图标，表示这个连接是加密过的：



公钥加密还有其他有趣的用途。例如，可以用做**数字签名**方案。设想张三想签署一条消息，以便让收信人能确定该消息是他本人签字认可，而不是别人假冒的。张三首先用自己的私钥对消息进行加密，然后把结果公布出去，然后任何人都能用张三的公钥来解密该消息。因为除了张三之外再也没有别人知道他的私钥，因此接收者可以断定

这条消息一定是张三加密过的。显然，这只有在张三的私钥没有泄露的情况下才成立。另外，张三还可以只给李四签署一条消息，让除李四之外的人都无法看到，而李四不仅能看到该消息，还能验证确实是张三发给他的。具体做法是：张三先用自己的私钥签署这条消息，然后用李四的公钥加密签署后的结果。王五虽然发现张三给李四发了一些信息，但只有李四能用自己的私钥解开密文，然后再用张三的公钥继续解密，同时证明了消息真是张三发来的。

当然，公钥加密方案也不是完美无缺的。如果张三的私钥泄露了，之前别人发给他的消息就形同明文，而他之前的签名也就不可信了。另外，尽管大多数密钥生成方案都包括密钥生成时间和失效时间，但真正撤销一个密钥（即宣布某个密钥从此以后失效），却是很困难的。

最常用的公钥加密算法是 RSA，这个算法是麻省理工学院的三位计算机科学家罗纳德・李维斯特（Ronald L. Rivest）、阿迪・沙米尔（Adi Shamir）和伦纳德・阿德曼（Leonard Adleman）在 1978 年发明的，RSA 就是这三位发明人姓氏的首字母缩写。RSA 算法的可靠性来自于对巨大合数做因数分解的难度。RSA 的工作原理是生成一个很大的整数（比如长度为 300 位），这个整数要有两个素因子（长度约为二者乘积的一半），以此作为生成公钥和私钥的基础。知道这两个素因子的人（即私钥的主人）能迅速解开密文，其他人要解密就得先分解素因子。由于计算量过大，这实际上是不可能完成的，或者最起码是我们认为无法完成的。李维斯特、沙米尔和阿德曼因为发明 RSA 而获得了 2002 年图灵奖。

在 RSA 算法中，密钥的长度很重要。据我们目前所知，分解大整数的运算量随整数长度的增长而呈指数增长。我曾经为了演示 RSA 而生成过 512 位密钥，这个长度也是 RSA 标准密钥生成程序的默认值。然而到了 2008 年左右，生成程序已经不让我用 512 位的密钥了，因为不够长，于是我用了 1024 位的。看样子，过不了几年，1024 位也要嫌短了。RSA 实验室是持有 RSA 专利的公司[①]，它曾举办过一个分解素因子大赛，

---

① 2000 年前，RSA 专利在美国有效，但在大多数其他国家都无效。由于专利问题，历史上曾经有很多涉及到数据加密的开源软件采取在美国之外运营、分别发布美国版和国际版、默认发行不包括 RSA 但允许用户从源代码自己编译等办法来规避在美国的专利限制，如 OpenBSD 操作系统 http://www.openbsd.org/27.html 。2000 年，RSA 专利在美国已到期。——译者注

从 1991 年一直办到 2007 年。它公布了一个列表，上面是越来越长的巨大合数，设立奖金悬赏能分解开每个合数的第一人。最小的合数有 100 多位，很快就分解开了。2007 年这个竞赛停办之时，分解出来的最大合数有 193 位（二进制 640 位），奖金是 2 万美元。（这个列表还挂在网上，感兴趣的读者不妨一试。）

现代密码术尽管有这些神奇的特性，但在实际应用时仍然需要一定程度的信任来支撑，特别是需要可信的第三方存在。举个例子，我在网上买书时，如何确定我是在和亚马逊对话，而不是和一个精于伪装的冒牌货往来？为了解决这个问题，我访问亚马逊时，该网站发送给我一个**证书**来证明它的身份。证书是由独立的权威机构签署的一组信息，用来证明亚马逊网站的身份。浏览器用权威机构发布的公钥来检查网站发来的证书，确认它属于亚马逊而不是其他机构。理论上，我可以相信：如果权威机构说它是亚马逊，那它一定就是。但我首先要信任权威机构本身，如果是权威机构是假冒的，那我就不能信任用它签署的任何网站证书。令人诧异的是，常见的浏览器包含了很多证书，比如我用的 Firefox 版本里有 80 多个，其中大多数来自我从没听说过的组织，比如中国台湾的中华电信公司和斯洛伐克的 Disig a.s.公司。2011 年 8 月，黑客入侵了荷兰的一家证书权威机构 DigiNotar，因而他就能生成包括 Google 在内的很多网站的假冒证书。

因为公钥算法很慢，所以一般不直接用公钥算法对文档签名，而是签署从原始文档提取出来的小得多的文档。选用适当方法生成的这种小文档无法伪造，也无法从其他文档生成同样的签名值。这样的小文档就叫做**消息摘要**或**密码学散列**，其创建方法是用某种算法把任意输入的比特流变换成固定长度的比特流，也就是摘要或散列，最终得到的结果具有如下特性：无法通过计算找到别的输入来生成同样的摘要。此外，输入中最轻微的改变都会让输出摘要中大约一半的比特发生变化。这样，通过比较接收者计算生成的文档摘要与原始摘要是否匹配，就能有效检测文档是否遭到了篡改。下面来看一下字母 x 和 X 的密码学散列值（用十六进制表示）：

```
x       9DD4E461268C8034F5C8564E155C67A6
X       02129BB861061D1A052C592E2DC6B383
```

这两个字母的 ASCII 值只差一位，散列值却大不相同。根本无法通过计算找到别的输入，使得输出的散列值和这两者中的一个相同，也无法从散列值反推回原始输入。

目前广泛使用的消息摘要算法有两个。一个是罗纳德 · 李维斯特开发的 MD5，生成 128 位的摘要。另一个是 SHA-1，来自 NIST（美国国家标准技术研究所），生成 160 位的摘要。已有研究表明 MD5 和 SHA-1 都有弱点[①]，因此不适合用于对安全性要求较高的场合。NIST 正在举办一场比赛，寻找新的算法 SHA-3 来解决 SHA-1 存在的问题[②]。

在任何安全系统中，最薄弱的环节都是作为用户的人，人会破坏那些他们觉得复杂难用的系统。设想一下，如果有人逼着你改密码，尤其是新密码必须马上想出来，而且要满足一些古怪的限制，比如要包含大小写字母、数字、某些特殊字符，但又不能包含另外一些字符，你会怎么做？你可能会像我那样用公式来生成新密码，或者把无规律的新密码写在纸上，但这两种做法都会带来潜在的安全风险。（问问你自己：如果攻击者得到你的两个密码，能不能猜出你的其他密码？）就算每个人都能坚守安全防线，志在必得的敌人还可以用收买、讹诈、盗窃、暴打这四大武器来攻破城防。而像政府这样合法的暴力机构，则越来越多地用牢狱之灾来恐吓那些不肯说出密码的人。

10.10　小结

1990 年以来，万维网从无到有，发展到如今已然无处不在了。万维网改变了商业，也改变了我们的很多习惯。尤其是在消费行为方面，过去谁能想到搜索、网上商店、评级系统、比价网站和产品试用网站的威力呢。

机遇和利益总是与问题和风险相逐又相随。万维网扩展了我们的生存空间，但也前所

---

① 根据抽屉原理，理论上，任何散列函数都必然存在碰撞，但一直以来，没有人成功构造出密码学散列函数的碰撞例子。2004 年，中国科学家王小云针对 MD5 等函数，成功构造出了这样的碰撞例子。随后，王小云又与其他科学家一起设计了计算量少于 263 步的 SHA-1 碰撞算法。——译者注

② 这场比赛已于 2012 年 10 月结束，四位科学家设计的 Keccak 算法获胜，但截至本书中文版定稿时，NIST 尚未公布 SHA-3 标准。考虑到 Rijndael 和 AES 的前例，最终的 SHA-3 标准可能会和 Keccak 有出入。SHA 算法家族还有 SHA-2 系列的 4 个算法，是 SHA-1 的变体，分别把摘要长度扩大到 224、256、384 和 512 位。SHA-3 不是为了取代 SHA-2，因为目前尚未出现对 SHA-2 的有效攻击。SHA-3 是为了寻找和 SHA-1 思路完全不一样的安全散列算法。参见 http://en.wikipedia.org/wiki/Secure_Hash_Algorithm 。——译者注

未有地让我们在陌生人面前暴露无遗，因此伤害也可能不期而至。

万维网引发了很多难解的司法管辖权问题。例如在美国，很多州对其境内的买卖征收营业税，但是亚马逊这样的网上商店，并不会对来自大部分州的买家收营业税，依据是他们在买家所在的州并没有实体商铺，因而无需帮那个州的税务局代为收税。按理说，买家应该自己上报外地购物记录并依法纳税，但没人这么做。

另外，诽谤罪的定罪也牵扯到司法管辖权。在有些国家，告某人诽谤只要能在该国看到载有相关言论的网站（无论网站在哪儿），就能胜诉，而被告是否到过这个国家都无所谓。

有些行为在某国合法，换一个国家可能就是非法的。常见的两个例子是色情业和在线赌博。一国公民在互联网上干了在本国犯法的事儿（比如到外国网站赌博），这个国家对他如何执法？有些国家仅为进出国门的互联网留下了有限的几条通路，以此来屏蔽他们不认可的网络访问，但这样做的国家在世界上也并非绝无仅有。对政府来说，还有一个办法是让人们用实名上网。这看上去是个防止匿名辱骂和骚扰的好办法，但却让持激进观点的人和异见人士发言时如履薄冰[①]。

个人、政府（不论是不是国民支持的）和企业（其利益往往跨越国界）之间总是充满了合法权益的纠葛。互联网则让这些问题更加错综复杂。

---

① 2007 年 7 月，韩国正式推行了网络实名制，要求后台实名，前台可以使用化名。但对 Twitter、Fackbook 等社交网络不作实名要求，理由是社交网站属私人领域，不适用实名制。2012 年 8 月，韩国宪法裁判所裁定网络实名制违宪，韩国广播通信委员会将按要求修改相关法规并废除实名制。

——译者注

第 11 章

数据、信息和隐私

数字技术为人类带来了无数便利，少了它，我们的生活会平淡乏味很多。与此同时，数字技术也给每个人的隐私带来了空前的威胁。而且，（用本书责任编辑的话来说）这种情况只会越来越糟糕。对个人隐私的这种侵害，有些是因为互联网及其应用，有些则是数字设备变得越来越小、越来越便宜、越来越快带来的副作用。与日俱增的处理能力、存储容量和通信带宽，使采集和保留各种信息、高效地分析数据，然后无远弗届地传播变得易如反掌，而且成本几乎可以忽略不计。

隐私常常就是安全的同义词。至少对每个个体而言，如果自己的生活信息被传播得随处可见，那怎么会让人感觉安全无忧呢？特别是互联网，它对个人安全已经产生了重大影响。这种影响更多体现在财务风险而非人身安全方面。因为互联网让人们从各种来源收集和整理信息变得异常容易，从而为电子入侵大开方便之门。

这一章，我们讨论几个有关隐私和安全的话题，跟大家讲讲应对措施，虽然长远来看无法完全避免，但至少可以知道如何降低风险。不过，安全毕竟是一个非常复杂的话题，所以本章只能粗略地讲个大概。而且，我们的内容也将主要围绕本书其他章节的话题展开，而不会在社会、经济、政治和法律等应对技术变革的这些方面着墨太多。

我看报纸上最近有一个广告，说人类将在接下来几年产生大约 2.5 泽字节的数据。很多人对泽字节没有什么概念，“泽”是 10^{21}，不管怎么说这都不是个小数字了。这么多的数据从哪来，利用它们能干什么？这个问题的答案耐人寻味，而且很可能隐藏着

很多严重的问题。因为其中有些数据虽然我们用不到，但却可能与我们息息相关。与我们相关的数据越多，陌生人掌握我们信息的可能性也就越大。

本章先从搜索开始讲，因为在我们使用搜索引擎寻找要浏览的网站时，收集数据的过程通常就开始了。从搜索开始，就会引出跟踪的话题，比如你访问了哪些站点，在浏览网站时都做了什么。然后，自然就是数据库的话题，涉及很多利益主体收集的各种数据。接着是数据汇总和数据挖掘，因为数据的价值只有在兼收并蓄且被找出规律的情况下才会显现出来。而这也正是隐私问题的高发地带，有关我们的多方的数据汇集起来，会让我们的隐私信息暴露无遗。接下来，我们聊聊为了享受娱乐或便捷服务，我们情愿告诉别人的个人信息。这些个人信息让汇总数据更容易也更具有商业价值，当然对我们隐私的威胁也更大。最后要讨论一下云计算。云计算公司在自己的服务器上提供存储和处理服务，而我们要把自己的数据完全交给这些公司。”

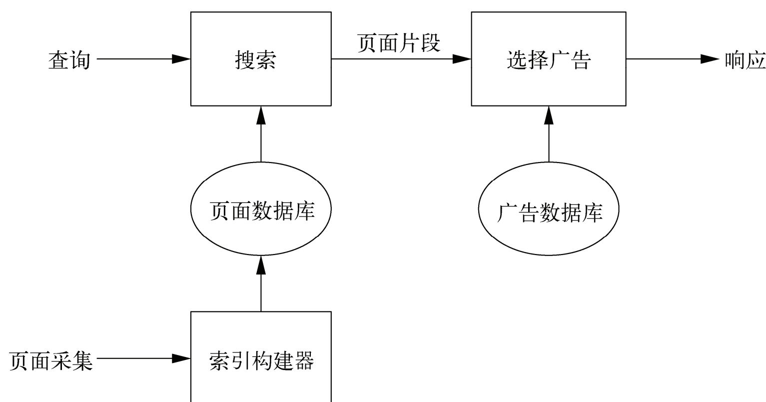
11.1 搜索

上网搜索的历史可以追溯到 1994 年前后。按照现在的规模来看，当时的 Web 渺小得几乎可以忽略不计。接下来几年间，网页和查询的数量均快速增长。谷歌公司的一篇论文（谢尔盖 · 布林和拉里 · 佩奇发表于 1998 年初的“The anatomy of a large-scale hypertextual web search engine”）中曾提到一个最成功的早期搜索引擎 Altavista，说它在 1997 年年底时每天要处理 2000 万个查询。这篇论文还准确地预言了 2000 年的 Web 将有 10 亿网页，每天要处理数亿次查询。《经济学人》2010 年底的一篇文章指出，仅谷歌公司一家每天就要处理 20 亿次搜索。

搜索是个大买卖，它从无到有，短短十年就成为一个产业。最典型的例子是谷歌，它成立于 1998 年，上市于 2004 年，到 2011 年上半年，市值已经达到 2000 亿美元。这个规模超过了很多传统的老牌公司。百年老店福特汽车公司的市值只相当于谷歌的四分之一，而赫赫有名的通用电气也只与谷歌打个平手。谷歌的创收能力超强，增长迅猛，但竞争对手林立，所以很难说将来会怎么样。（在此必须得声明一下：我有时候也会为谷歌提供咨询服务，甚至做它的兼职。显然，本书中的任何观点都只代表我自己，与谷歌公司无关。）

搜索引擎怎么工作？从用户角度来看，他们会在网页上的表单中填写查询条件，然后把这个条件发送给服务器。服务器差不多马上就会返回一组链接和文本摘要。从服务器的角度看则要复杂得多。服务器会生成一组包含查询关键词的页面，按照相关程度进行排序，再在 HTML 中附上页面的摘要，然后再发给用户。互联网实在太大了，每个用户在查询时不可能直接搜索整个互联网。搜索引擎的主要任务就是为满足查询需求，事先把所有网页的副本有组织地保存在服务器上。为此，搜索引擎需要利用爬虫程序采集互联网上的所有页面，把它们的内容保存到数据库中，以便后续查询可以迅速返回结果。（并不是所有人都明白搜索结果是以预先计算好的缓存页面的索引为依托的。就在几年前，美国一家国家级电视台的新闻节目称："输入'帕丽斯·希尔顿'，谷歌的计算机就会搜索整个互联网，只要半秒钟就能找到所有提到她的网页和她在网上的所有照片。"）

下面这张示意图可以大致说明搜索引擎的工作过程，包括在结果页面中插入广告：



关键是规模太大。用户几十亿，网页不知道有多少个几十亿。谷歌以前总爱公布它为构建索引采集了多少多少页面，但在这个数字超过 100 亿之后，就没见他们再公布过。如果一个网页平均 10 到 20 KB，那么存储 500 或 1000 亿页面就要占用上拍字节的磁盘空间。虽然有些页面是静态的，也就是说几个月甚至几年都不会更新，但也有相当多的页面更新得非常快（股票报价、新网站、博客、Twitter 消息等），因此爬虫的采集工作一刻也不能停。一停，索引信息就面临过时的风险。搜索引擎每天要处理几十亿次查询，每次查询都必须扫描数据库、搜索相关页面、按一定规则排序。在此期间

还要精心选择与搜索结果匹配的广告，在后台把整个过程涉及的一切都记录下来，从而进一步改进搜索质量，领先于竞争对手，卖出更多广告。

从本书的角度来说，搜索引擎是实际应用算法的一个典型案例。但巨大的通信量决定了简单的算法难以满足高性能的需求。

仅仅是与采集网页相关的算法就有一整套。有的用于判断下一次采集哪个页面，有的用于从网页中提取可供索引的信息（词、链接、图片等），有的用于把这些信息发送给索引构建器。首先从提取 URL 开始，然后剔除其中重复或无关的，剩下的才会添加到采集队列中。采集过程中比较棘手的问题在于不能过于频繁地爬一个站点，否则会显著增加站点负载，没准还会惹恼网站所有者（最终导致爬虫被拒之门外）。由于页面变化的速度千差万别，因此算法必须准确判断页面变化的频率，从而保证对变化快的站点提高采集频率。

采集来页面之后就要构建索引。这个阶段要从爬虫采集的页面中提取要索引的页面内容，然后在索引中记录这些内容，以及它们在页面中的位置和相应的 URL。这一阶段的具体处理方式取决于要索引的内容是文本、图片，还是 PDF 等标准文档格式。本质上，索引就是为网页中出现的词或其他可索引项创建一组页面和位置，并以相应的方式保存起来，以便通过任何具体的关键词都能够迅速地找到它们。

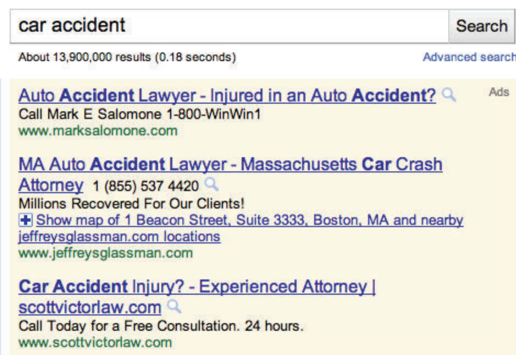
最后一步就是针对具体的查询组合响应页面。简单来讲，就是以查询中所有的关键词为依据，通过索引列表迅速找到匹配的 URL，然后（同样也是迅速地）选择匹配度最高的 URL。这个过程涉及的技术是搜索引擎运营商的核心竞争力所在，在网上不大可能搜索到相关的技术文档。同样，规模过大仍然是问题的关键。任何一个关键词都可能出现在数百万个页面中，两个关键词同时出现在上百万个页面中的概率也很高。关键在于怎么才能迅速从这些页面中筛选出最为匹配的十个页面。谁能把最佳匹配结果排在前头，谁的响应速度快，谁就能赢得用户。

第一批搜索引擎只会显示一组包含搜索关键词的页面，而随着网页数量激增，搜索结果中就会混入大量无关页面。谷歌的 PageRank 算法会给每个页面赋予一个权重，权重大小取决于是否有其他页面引用该页面，以及引用该页面的其他页面自身的权重。从理论上讲，权重越大的页面与查询的相关度就越高。正如布林和佩奇所说："凭直

觉，那些经常被其他网页提及和引用的页面的价值一定更高一些。”当然，要产生高质量的搜索结果绝对不会只靠这一点。搜索引擎公司会不断采取措施来改进自己的结果质量，以期超越对手。

面对如此巨大的数据量，要提供搜索服务也必须拥有庞大的计算资源。处理器要数百万，内存要以太字节计，硬盘要以拍字节计，带宽要达到每秒数吉比特，耗电量也要数十亿瓦。当然，还需要大量的人。这些投入都要花钱，而钱通常来自广告收入。简单来说，搜索引擎的广告模式是这样的。广告客户付钱在网页上显示广告，价格由多少人看过以及什么样的人看到该网页来决定。这种定价模式叫按页面浏览量收费，即按“展示”，也就是按广告在页面上被展示的次数收费。另一种模式是按点击收费，即按浏览者点击广告的次数收费。对广告感兴趣的浏览者无疑是有价值的，因此搜索引擎的广告模式，说到底就是拍卖搜索关键词。广告客户出价购买的是在特定关键词的搜索结果旁边显示广告的权利，而浏览者点击了广告，搜索引擎公司就会向广告客户收钱。

举个例子，假设有人搜索“车祸”，那这个人很可能是想找一位熟稔责任界定的律师。如果知道搜索人的地理位置信息，那么结果就会更精准，因为律师通常都希望本地人能看到自己的广告。同样，如果知道搜索人的其他一些情况，比如性别、婚否和年龄，那么他的搜索关键词就会更有价值。下面是我在马萨诸塞州坎布里奇（Cambridge, Massachusetts）搜索“car accident”（车祸）得到的结果。仔细看一看，尽管我没有提供地理位置信息，但搜索引擎是大概知道这个信息的。



最前面的搜索结果（不算广告）与防止发生交通事故有关，可见搜索结果并没有倾向于广告客户，这些人基本都是律师。（右侧的广告看起来跟请律师也没关系，但只要

我不点它，广告客户就不会花钱。）如果我点了其中一个广告，投放该广告的人就要根据自己的出价付钱给谷歌。搜索引擎公司都有完备的手段避免虚假点击，这其中的技术细节同样是秘而不宣的，特别是顾及到很多虚假点击都由僵尸网络产生的背景，这个问题我们在上一章里讲到过。

谷歌的 AdWords 简化了在线投放广告的评估过程，比如其估算工具会告诉你关键词“kernighan”的费用大约为每次点击 65 美分。换句话说，每当有人搜索 `kernighan` 并点击了我的广告（应该是最上面的两个结果中的一个），那我就得付谷歌 65 美分。这个工具还会估计出每天可能有 600 次搜索，当然谁也不可能知道有多少人会点击我的广告，我因此一天要花多少钱。我从来也没做过这个试验，因此结果至今还是未知数。

广告客户可以花钱让搜索结果倾向自己而不是竞争对手吗？布林和佩奇同样担心过这个问题，他们曾写道：“我们认为靠广告支撑的搜索引擎会本能地偏向广告客户，而远离消费者需求。”谷歌的大部分收入来自广告，但该公司会严格区分搜索结果和广告，其他主流搜索引擎也会这么做。

我们这里讨论的虽然是搜索，但同样的考量对任何广告行为也都适用：定位越精准，效果越明显。

11.2　跟踪

只要上网，我们的信息就免不了被收集。不留下蛛丝马迹，几乎什么也干不了。使用其他系统时也一样，特别是使用手机的时候，手机网络随时都知道我们的位置在哪里。如果是在户外，支持 GPS 的手机（现在的智能手机几乎都支持）定位用户的误差不超过 10 米，而且随时都会报告你的位置。（我在办公室时，我的手机报告我的位置误差也只有 20 米，而办公室所在的大楼属于中等规模建筑。）有些数码相机也带 GPS，可以在照片中编入地理位置信息，这种做法被称为**打地理标签**。

把多个来源的跟踪信息汇总起来，就可以绘制一幅关于个人的活动、喜好、财务状况，以及其他很多方面的信息图。这些信息最起码可以让广告客户更精准地定位我们，让我们看到乐意点击的广告。不过，跟踪数据的应用可远不止于此。这些数据还可能被

用在很多我们意想不到的地方。比如根据收入把人分成三六九等，在贷款时区别对待，或者更糟糕地，被人冒名顶替，被政府监控，被人谋财，甚至害命。

怎么收集我们的浏览信息呢？有些信息会随着浏览器的每一次请求发送给服务器，包括你的 IP 地址、正在浏览的页面（即来源页）、浏览器的类型和版本、操作系统，还有语言偏好。

此外，如果计算机中保存着服务器域的 cookie，那么这些“小甜饼”也会随浏览器请求一块发送。根据 cookie 的规范，只能把这些保存用户信息的小文件发给最初生成它们的域。那还怎么利用 cookie 跟踪我对其他网站的访问呢？

要知道答案，就得明白链接的工作原理。每个网页都包含指向其他页面的链接（这正是“超链接”的本义）。我们都知道链接必须由我们主动点击，然后浏览器才会打开或转向新页面。但图片不需要任何人点击，它会随着页面加载而自动下载。网页中引用的图片可以来自任何域。于是，浏览器在取得图片时，提供该图片的域（根据请求中的来源页信息）就知道我访问过哪个页面了。而且这个域也可以在我的计算机上存放 cookie，并且收到之前访问该域时生成的 cookie。

以上就是实现跟踪的秘密所在，下面我们再通过例子来解释一下。假设我想买一辆新车，因此访问了 toyota.com。我的浏览器因此会下载 60 KB 的 HTML 文件，还有一些 JavaScript，以及 40 张图片。其中一张图片的源代码如下：

```
<img src="http://ad.doubleclick.net/ad/
          N2724.deduped_spotlight/B1009212;
             sz=1x1;tag=total_traffic;ord=1?"
    width=1 height=1 border=0>
```

这个`<img>`标签会让浏览器从 ad.doubleclick.net 下载一张图片。这张图片的宽和高都只有 1 个像素，没有边框，而且很可能是透明的，总之页面上看不见它。当然，这张图片根本就没想让人看到。当我的浏览器请求它时，DoubleClick 会知道我正在浏览丰田汽车公司网站的某个页面，而且（如果我允许）还会在我的计算机中保存一个 cookie 文件。要是我随后又访问了一个内置 DoubleClick 图片的网站，DoubleClick 就可以绘制一张我的“足迹图”。如果我的“足迹”大都留在汽车网站上，DoubleClick 会把这个信息透露给自己的广告客户。于是乎，我就能看到汽车经销商、购车贷款、修车服

务、汽车配件等等各种广告。如果我的“足迹”更多与交通事故或止疼有关，那么就会看到律师和医生投放的广告。

拿到用户访问过的站点信息后，DoubleClick（现为谷歌所有）会根据这些信息向丰田等广告客户推销广告位。丰田公司继而利用这些信息定向投放广告，而且（可能）会参考包括我的 IP 地址在内的其他信息。（DoubleClick 不会把这些信息卖给任何人。）随着我访问的页面越来越多，DoubleClick 就可以绘制出一幅关于我的形象，借以推断我的个性、爱好，甚至知道我已经 60 多岁了，性别男，收入中上，住在新泽西中部，就职于普林斯顿大学。知道我的信息越多，DoubleClick 的广告客户投放的广告就越精准。到了某个时刻，DoubleClick 甚至可以确定那个人就是我。尽管大多数公司都声称不会这么做，可假如我的确在某些网页中填过自己的名字和电子邮件地址，那谁也不敢保证这些信息不会被传播得到处都是。

这套互联网广告系统设计得极其精密。打开一个网页，这个网页的发布者会立即通知雅虎的 Right Media 或谷歌的 Ad Exchange，说这个网页上有一个空地儿正虚位以待，可以显示广告。同时发过去的还有浏览者的信息（例如，25 到 40 岁之间、单身女子、住在旧金山，是个技术宅，喜欢泡馆子）。于是，广告客户会为这个广告位而竞价，胜出者的广告将被插入到这个网页中。整个过程只有零点几秒。

有不少公司在做收集网民“足迹”的业务，不过这个行业也在整合。DoubleClick 原先是一家独立公司，后来在 2008 年被谷歌花 31 亿美元收购了。最近有一篇论文提到，一半以上最受欢迎的网站都开启了跟踪机制。有些跟踪技术比所谓的“第三方”cookie（即像 DoubleClick 保存到本地计算机上的与网页不在同一个域的 cookie）更难觉察，也更难禁止。许多网站都含有多家公司的跟踪程序，我前些天访问的一个网站就有 12 个。给大家推荐一个浏览器扩展 Ghostery，通过它可以禁用 JavaScript 跟踪代码，还能查看被阻止的跟踪器。装上它，你会惊讶于互联网上潜伏着多少“间谍”。

要是你不喜欢被跟踪，完全可以把它们纠出来制服，或者至少暂时甩开它们。当然，这样做会耗费你的一些精力和体力。浏览器其实允许你完全关闭 cookie，或者禁用第三方 cookie，或者在浏览器关闭时自动把 cookie 删除。长时间保存的 cookie 会存储在计算机的文件系统中。若要删除它们，可以通过浏览器中的某个按钮，也可以直接找到并删除那些保存它们的文件。大多数公司的跟踪程序都支持自愿回避

（opt-out）机制。也就是说，如果跟踪程序在你的计算机中碰到了一个特殊的 cookie，那它们就不会再为了定向广告而记录你的行踪，但还是有可能记录你的 IP 地址。我在使用 Firefox 的一个扩展 TACO（Target Advertising Cookie Opt-out，定向广告 Cookie 自愿回避），这个扩展维护着一个 cookie 跟踪站点的列表（目前有大约 150 个名字），在浏览器中保存着它们的自愿回避 cookie。而我呢，同时对大多数网站都选择关闭 cookie。

这些机制在不同的浏览器中并不兼容，甚至同一款浏览器的不同版本之间都不兼容，而浏览器默认的设置通常是允许 cookie 的读写。

封掉第三方 cookie 有时候也没用，因为浏览器中显示的网页好像是来自一个站点，但实际却是从其他站点转发过来的。比如下面这个假想的 URL：

```
http://www.bigcorp.com/relay?doubleclick.net/whatever
```

它首先会被 BigCorp 域中的服务器解析，然后转给 DoubleClick，而后者仍然会提供相同的 cookie 信息，通过 BigCorp 服务器加载到你的浏览器中。这种技术实际上是把第三方 cookie 打扮成了第一方 cookie。

遗憾的是，很多站点离开 cookie 都无法运行。当然有时候是合情合理的，比如服务器需要知道你是不是已经登录过了，而有时候则就是想要跟踪你。这让我很恼火，所以我一般对这样的网站都敬而远之。

前面介绍过，一个像素大的图片或者叫网页信标（web beacon）也可以用来跟踪你。用于取得像素图片的 URL 可以包含一个标识码，表示你正在浏览什么网页，还可以包含一个标识符，表示特定的用户。这两个标志就足以跟踪你的浏览活动了。

互联网视频网站普遍使用的 Adobe Flash 技术能够实现动画广告，该技术也会在你的计算机中保存 Flash cookie。这种 cookie 与前面所说的 cookie 不是一回事，在文件系统中跟它们也不是保存在一个地方。Flash cookie 本来是用于缓存数据以便更流畅地播放视频，但也会被用于跟踪。Adobe 提供了禁用它们的方法，为此需要访问 adobe.com 并完成几个页面的设置。说来也怪，我发现关闭 Flash cookie 并不影响性能（对每台新电脑我都会这么做）。换句话说，这种缓存好像没什么效果。

JavaScript 也是常用的一种跟踪技术。不管是包含在 HTML 中的 JavaScript 脚本代码，还是通过`<script>`标签中 `src="name.js"`属性的 URL 下载的脚本文件，浏览器都会立即执行。JavaScript 最主要用于“分析”用户浏览特定网页的行为。比如，下面这行代码会从谷歌服务器加载并运行一段 JavaScript 脚本：

```
<script type="text/javascript"
    src="http://pagead2.googlesyndication.com/pagead/show_ads.js">
</script>
```

JavaScript 代码可以读写 cookie，有时候还可以读写浏览器访问过哪些页面的历史记录。它还可以持续监视鼠标在屏幕上的位置，并将这些信息发回服务器，以便推断网页的哪些部分更吸引用户，哪些部分用户很少关注；还可以监视鼠标点击了哪些位置（即使这些位置不是链接等能够作出反应的区域）。现在，偷偷摸摸使用 JavaScript 的情况有越来越多的趋势。如果网页中有 JavaScript，那么发生恶意攻击的可能性也会增大。

2010 年底有一篇文章谈到 evercookie，文中列举了在客户计算机上保存跟踪信息的十几种可能的方式[①]，有些已经超出了浏览器的控制范围。就算是这方面的专家，想把它们都清除干净也很困难。而只要有一个地方有残留，其他地方的跟踪功能就可以全部恢复。很快，有关防护措施的文章也见诸报端，但其揭示的问题确实无法否认：除了得到人们正式认可的地方之外，还有很多地方可以保存这些信息。比如，有的跟踪手段是收集某人使用的操作系统、浏览器、版本号、语言偏好、安装的软件等特征，这些都与 cookie 无关。收集到足够的特征信息后，就有可能据以区分和识别某个人。广告客户和其他机构对这种精确的身份识别自然是欣赏有加的。

并不是只有浏览器可以跟踪用户，邮件阅读器、视频播放器等等使用 HTTP 和 HTML 的软件也可以。如果你的邮件阅读器解析 HTML，那它就有可能“显示”那些 1 个像素大的图片，于是你就被别人给跟踪了。

一个为害特别大的跟踪和监视手段时不时就会浮出水面，它就是**深度包检测**（deep packet inspection）。前面不是说过嘛，每个 IP 数据包从你的计算机出发，通常都要经过 15~20 个网关才能到达目的地，反过来也差不多。这条路径上的每个网关都有可能检查这些数

---

① 参见 http://en.wikipedia.org/wiki/Evercookie。——译者注

据包，看到里面的信息，甚至还能以某种方式修改它们。一般来说，这种入侵都是你的ISP发起的，因为在那里最容易认出你来。广告客户和政府部门尤其喜欢这种信息，因为这些信息已经超出了上网浏览的范畴，而是涉及你在互联网上的一举一动。

关于个人身份的哪些信息可以收集，可以怎么使用，不同国家有不同的规定。以美国为例，简言之，怎么都行！任何公司或机构都可以收集和传播关于你的信息，不用通知你，也不需要给你提供自愿选择的机会。而在欧盟（同样简言之），隐私是一个严肃得多的话题：在没有得到个人明确许可的情况下，任何公司都不能收集或使用个人数据。

谷歌街景（Street View）的不同遭遇，就足以表现出这种法律和文化上的差异。谷歌的街景服务在世界很多地方提供街道的全景照片。而对所有照片，谷歌都会通过算法来模糊人脸以防这个人被认出来，同样也会模糊车牌。尽管如此，它在很多国家还是引发了隐私问题。但这些国家在看待街景服务的价值和它暴露个人隐私的风险上所采取的立场却又各不相同。

11.3　数据库、信息与聚合

互联网和Web已经彻底改变了人们收集、存储和展现信息的方式。搜索引擎和数据库对每个人都具有不可估量的价值。很难想象之前没有互联网的时代我们是怎么过来的。凡事都有两面，现在这样数据在网上随意传播也有问题，尤其是那些可能会过多暴露我们的信息如果传出去，会令人相当不自在。

有些信息明显就是公开的，还有些信息收集起来就是为了供人搜索和索引的。如果我写了一个网页，希望大家都能看到，假设就是这本书的页面吧，那么我肯定愿意人们通过搜索引擎可以轻易发现它。

那怎么看待公共档案呢？法律上，某些信息属于“公共档案”（public records），任何人通过申请都可以查阅。在美国，公共档案包括可以公开的庭审记录、抵押文件、房价、地方房产税、出生和死亡记录、结婚证、政治捐助，等等。（查阅出生记录通常是为了知道“妈妈婚前的姓氏”，以便辅助确认一个人的身份。）很早以前，要知道这些信息必须不辞劳苦，亲自前往当地政府驻地查阅。因此，虽然这些档案名义上是“公开”的，但不付出点代价也不可能看到。谁要想获得这些数据，就得亲自跑一趟，或

许需要出示身份证件，要想复制一份可能还得花点钱。今天，如果这些数据上了网，我坐在自己家里就可以轻轻松松查阅这些公共档案。我甚至可以开个公司，收集汇总这些信息，然后与其他信息整合起来。比如很多人都知道的 zillow.com，就整合了地图、房地产广告、有关财产和交易的公开数据，通过地图来直观地显示房价。如果你想买房或者想卖房，它对你了解市场很有用；否则，你可能会觉得它暴露了人家太多的信息。通过查询联邦选举委员会（FEC，Federal Election Commission）的选举捐款数据库（fec.gov），可以知道哪位候选人得到哪些朋友和要人的捐赠，或许可以查到他们的家庭住址等信息。在 FEC 提供信息的基础上，fundrace.huffingtonpost.com 在一张地图上给我们标出了这些人的名字、地址、职业。这种做法让人们对如何平衡公众知情权和个人隐私权有了新的认识。

什么样的信息才应该让人如此轻而易举地得到？这个问题很难回答。政治捐款应该公开，但门牌号码可能就应该稍加隐藏。包含美国社会保险号等个人身份识别信息的公共档案似乎不该放在网上，因为这就给盗用别人身份打开了方便之门。可当前的法律无法完全阻止这种信息的公布，而这种信息一旦上网，就覆水难收了。

随着在多个各不相关的来源都能查到同一类信息，这个问题就变得愈发严重了。比如，很多提供 Web 服务的公司都有自己大量的客户信息。搜索引擎会记录所有查询，也包括查询人的许多信息。最低限度也会记录查询人的 IP 地址，还有用户之前访问过网站时保存在计算机上的 cookie。

2006 年 8 月，AOL 出于好意而公开了一大批查询日志样本，供人研究。这些日志涉及三个多月以来 65 万用户的 2000 万查询，已经做了匿名处理，因此从理论上讲，不存在任何可以用于辨识个人身份的信息。尽管是善意之举，但人们也很快就发现这些日志在实践中不会像 AOL 想象的那样做到完全匿名。每个用户在查询时都会被赋予一个随机但唯一的标识符，有了这个标识符，就很容易知道同一个人都查询过什么内容。进而，确定一些人的身份也就成为可能。因为不少人都搜索过自己名字、地址、社会保险号以及其他个人信息，通过搜索相关性分析暴露出来的信息比 AOL 认为的多，也肯定比原始用户自己想到的多得多。AOL 很快从自己网站上删除了这些日志，当然为时已晚。这些数据早已被传播得满世界都是了，而且至今仍可以找到，甚至还附有帮你分析它们的一些工具。

查询日志对经营企业和改进服务有价值，但很明显其中可能包含敏感的个人信息。谷歌、雅虎、微软这些提供搜索服务的公司会把查询日志保留多长时间？这里有个矛盾：考虑个人隐私则保留的时间应该短，而考虑执法目的则保留的时间应该长。为了达到一定的匿名程度，这些公司内部该对数据进行怎样的处理？虽然他们全都声称会删除每条查询对应 IP 的部分信息（一般是最右边那一字节），但仅仅如此似乎还不够，还达不到反识别用户的目的。政府机关查询这些信息的权限有多大？打一次官司会查询多少信息？所有这些问题都没有明确的答案。AOL 公布的查询日志中有些是很吓人的，比如有人查询怎么杀死自己的配偶。因此，有限度地向司法机关开放这些数据是合理的，但问题是这个限度应该放多大，很难说清楚。

AOL 事件揭示了一个广泛存在的问题，即真正做到数据匿名化是非常困难的。删除身份识别信息可以降低识别度，单就特定的数据而言，确实无法定位到用户，因此可以说它是无害的。但现实当中信息的来源是多方面的，把多个来源的信息组合起来则很可能挖掘出更多身份特征。而且某些来源的信息甚至连提供者自己都不知道，这些信息将来也未必还能找得到。举个例子，假设搜索引擎会删除每条查询对应 IP 的最右边一个字节，但根据剩下的三个字节仍然可知它来自普林斯顿大学计算机科学系，如果再结合普林斯顿日志中我什么时候使用过该 IP 上网的记录,那就可以把具体的查询跟我挂上钩了。

有关这种再识别（re-identify）问题，下面可以给大家讲一个真实的案例。1997 年，当时在 MIT 读博士的拉坦娅・斯威尼（Latanya Sweeney）分析了马萨诸塞州 135 000 名雇员的体检记录，这些记录都做了反识别处理。数据来源是该州的保险委员会，可用于研究目的，甚至被卖给了私人公司。每条体检记录中除了大量其他信息外，都包括生日、性别和邮政编码。斯威尼发现有 6 个人的生日都是 1945 年 7 月 31 日，其中 3 个男性，而只有 1 人住在坎布里奇。把这些信息和公开的选民登记名单一对照，她便知道了这个人就是时任州长威廉・韦尔德（William Weld）。

匿名处理数据与混淆保证安全（前一章刚介绍过）多少有些类似之处，这两者都是基于没有足够信息无法解密数据的考虑。问题是，这两种情况下敌人掌握的信息，很可能比我们想象的多。而且就算眼下他们不知道，将来也有可能知道。

11.4　隐私失控

不久前，我在网上看到一篇文章，大概是这么写的："有一次面试，他们问了一些我简历上没写的问题。原来他们看了我的 Facebook 主页，这太让人意外了。Facebook 上可都是我个人隐私啊，跟他们有什么关系！"这个人很傻很天真，但我想很多 Facebook 用户在这种情况下可能都会有一种被冒犯的感觉，尽管他们清楚地知道公司人力资源部和大学招生办会例行通过搜索引擎、社交网站及其他类似工具来了解申请人的更多信息。在美国，面试时问一个人的民族、宗教信仰、性取向等很多关乎个人的问题都是非法的，但这些问题通过社交网站和搜索引擎都可以不费吹灰之力就找到答案。

最重要的是要知道，跟踪我们浏览的网站只是收集我们信息的诸多方式中的一种。毋庸置疑，随着社交网站的流行，为了娱乐和与其他人联系，我们**自愿**放弃了很多个人隐私。

社交网站存在隐私问题是毫无疑义的，因为它们会收集注册用户的大量信息，而且是通过把这些信息卖给广告客户来赚钱。尽管出现的时间不长，但它们的用户规模增长迅猛。Facebook 成立于 2004 年，现在据说已经有了 7.5 亿用户，相当于全世界人口的十分之一还多。如此之快的增长速度，不可能有太多时间考虑隐私政策，也不可能从容不迫地开发出稳定可靠的计算机程序。于是，每个社交网站都面临着因功能不完善而泄露用户隐私、用户不清楚该如何选择自己的隐私设置（变得太快）、软件出错，以及由于系统固有问题而暴露数据等问题。

作为最大也最成功的社交网站，Facebook 的问题也最明显。Facebook 给第三方提供了 API，以方便编写 Facebook 用户可以使用的应用。但这些 API 有时候会违背公司隐私政策透露一些隐私信息。当然，并非只有 Facebook 一家如此。做地理定位服务的 Foursquare 会在手机上显示用户的位置，能够为找朋友和基于位置的游戏提供方便。在知道潜在用户位置的情况下，定向广告的效果特别好。如果你走到一家餐馆的门口，而手机上恰好是关于这家餐馆的报道，那你很可能就会推门进去体验一下。虽然让朋友知道你在哪儿没什么问题，但把自己的位置昭告天下则非明智之举。比如，有人做了一个示范性的网站叫"来抢劫我吧"（Please Rob Me），该网站根据 Foursquare 用户在 Twitter 上发表的微博可以推断出他们什么时候不在家，这就为入室行窃提供了机会。

“位置隐私”——保证自己位置信息保密的权利——在我们日常使用的很多系统中并没有得到保障，比如信用卡支付系统、高速公路和公交车刷卡收费系统，当然还有手机网络。要想让人对你都去过哪儿一点都不知情越来越困难。手机应用经常会要求访问你在手机上存储的一切信息，包括通话记录、本地存储的信息、当前位置，等等。在我看来，这些应用想知道的已经超出了它们应该知道的。

社交网站和其他一些站点甚至会泄漏非用户的个人信息。举个例子，假如一位好心的朋友发给我一份电子请柬（e-vite），请我去参加某个聚会。就算我不答复这个邀请，也没有允许别人使用我的电子邮件地址，运营该邀请服务的公司就已经得到了我准确的电子邮件地址。如果一位朋友从他的 Gmail 或雅虎账号给我发了封邮件，那么我的邮件地址就在没得到我许可的情况下被别人知道了。如果一位朋友在一张照片中给我打上标签，然后将它发布到 Facebook 或 Flickr（或两个地方都发），那我的隐私就在没有我同意的情况下暴露了。Facebook 有图像识别功能，因而那位朋友在给我加标签时会更方便，而且这个操作默认无需经过我这个被标签人同意。所以说，社交网站很容易根据自己的用户构建一个交往群体的“社交图谱”，其中包括被这些用户牵连进来但并未同意甚至毫不知情的人。在以上几种情形下，任何人都束手无策，而且在自己的信息公开后也没有办法删除它们。

情报机关早就知道通过流量分析来了解大量内幕消息，只要知道谁跟谁有联系即可，都不用知道当事人说了什么。同样，通过人们在社交网站或明或暗的联系也可以掌握很多“情报”。比如，2009 年两名 MIT 学生声称可以根据人们在 Facebook 上朋友的性取向推断出这些人的性取向。无论能否准确推断出某个人的性取向，但至少这种推断是可行的。可以肯定的是，美国政府早已着手挖掘异议人士在 Facebook 网页上的信息，借以了解还有谁跟他们“同流合污”。

11.5 云计算

大家先回忆一下第 6 章介绍的计算模型。相信你至少有一台电脑，也许更多。在电脑上，不同的任务需要使用不同的软件来完成，比如用 Word 来创建文档、用 Quicken 或电子表格软件记账、用 iPhoto 或 Picasa 管理照片。虽然这些软件在你电脑上运行，

但它们有可能会上网使用某些服务。你可能经常要下载这些软件的一些补丁以修复 bug，偶尔还会为了得到某些新功能而购买它们的升级版。

这种计算模型的本质是程序及其数据都保存在你的计算机中。如果你在公司或在路上，突然需要一个保存在你家里计算机中的文件，那对不起，没办法。如果你想在 PC 和 Mac 上都使用 Excel 或 PowerPoint，那就得一台机器买一个。如果你在一台计算机上修改了文件，然后想在另一台计算机上使用该文件，那得自己想法办拷过去。

与此同时，另外一种计算模型越来越普及，那就是使用浏览器来访问和操作在互联网服务器上保存的信息。Gmail 和雅虎等邮件服务是比较常见的例子。任何计算机，甚至很多手机都可以收发邮件。当然，把在本地写的邮件上传到服务器，或把邮件下载保存到本地文件系统都没问题，但这不是必须的。更多的时候，只要把邮件保存在服务器上就行了。不用考虑升级软件，但新功能照样不期而至。而且这些服务几乎全都是免费的，唯一的“成本”就是在你看邮件的时候瞄两眼广告。当然，你也可以像我一样，对广告视而不见。（偶尔我也会看一看广告，但通常都会觉得莫名其妙。比如，我收到朋友的邮件，感谢我阅读了一份手稿，同时显示的广告则是关于心理咨询的，很明显是因为邮件里包含“阅读”两个字。）

这种模型通常被称为**云计算**，也就是把互联网比喻成了不会固定在某个地理位置的“云”，而信息都保存在“云端”。邮件是最常见的云服务，但其他类型的也非常多。比如，Flickr、Dropbox、在线日历和社交网站。数据不保存在本地，而是保存在云上。换句话说，就是保存在服务提供商的服务器上，比如 Gmail 邮件保存在谷歌服务器上，照片保存在 Flickr 服务器上。云计算模型下的软件通常是由本地部分（在客户端运行）加云端部分（在服务器端运行）构成，但都是从云下载到浏览器的。

云计算成为现实有赖于多方面条件的成熟。个人计算机能力越来越强，浏览器也一样日新月异。现在的浏览器已经可以高效地运行涉及大量显示处理的大型程序，而编程语言使用的仍然是解释型的 JavaScript。与 10 年前相比，服务器到客户端的带宽和延迟有了明显改善。这就为快速地发送和取回数据提供了保障，就算是响应一次次的键盘敲击都不成问题。于是，原先必须依靠独立的软件才能处理好的大部分用户交互功能，现在通过浏览器就可以搞定，而且可以把大量数据保存在服务器上。

以浏览器为运行环境的应用，其响应速度几乎与桌面应用一般无二，但却可以访问保存在任何地方的数据。举一个最有代表性的例子吧。Google Docs 就是一个云办公系统，包括文字处理、电子表格和 PPT 程序。至少粗略来看，它已经完全可以代替 Microsoft Office，而且支持随处访问和多用户同步更新等功能。这里面最有意思的问题是，云办公系统最终能否抢滩桌面版。不用说，微软最关心这个问题，因为其 Office 产品收入在总收入中占有相当份额。而且，Office 主要在 Windows 上运行，而 Windows 又是微软另一个主要收入来源。浏览器版的文字处理和电子表格应用不依赖微软的任何产品，因而直接威胁这两块核心利益。目前，Google Docs 及类似的系统还不具备 Word 和 Excel 的全部功能。然而，纵观技术发展史，明显不如前一代产品完善的新产品，从更完善的老产品那抢夺用户，逐渐将其挤出市场，这样的事例俯拾皆是。（克莱顿・克里斯坦森的《创新者的窘境》一书对此有深刻的分析。）微软显然对这些程序的云端版非常在意，而且事实上也提供了 Office 的在线版。

云计算依赖客户端的快速处理和大量内存，以及服务器端的高带宽。客户端代码用 JavaScript 编写，通常会非常复杂。JavaScript 代码无疑会要求浏览器快速刷新图形显示，并对用户的操作（如拖放）和服务器端操作（如更新内容）迅速作出响应。要做到这一点很困难，而浏览器与 JavaScript 版本的不兼容则雪上加霜。为此，开发人员必须知道如何有效地把正确的代码传送给客户端。不过，这两方面的问题都会得到解决，因为计算机越来越快，而标准也越来越得人心。

云计算可以转移计算的负载，转移处理过程中数据存储的位置。比如，为了让 JavaScript 代码适应更多的浏览器，可以在代码中使用条件判断，类似于“如果当前浏览器是 Internet Explorer 8，则执行如下代码；如果是 Safari 5，则执行如下代码；否则，执行如下代码”。这种代码经常一写就是一堆，而代码一多，就要占用更多服务器到浏览器的带宽。同时，一次次的检测也会导致浏览器运行变慢。换一种思路，可以把负载转移到服务器。让服务器询问客户端正使用什么浏览器，然后只将适合该浏览器的代码发过来。这样可以保证代码尽可能少，运行尽可能快（当然，程序规模不大时，差别可能不明显）。

网页内容可以不压缩，这样前后端处理速度都很快，但占用带宽多。如果压缩，占用的带宽少，但会影响前后端的处理速度。有时候，只在一端压缩也可以。比如对大型

JavaScript 程序，开发人员经常会去除所有不必要的空格，使用两个字母来代替变量和函数名，以此来压缩代码。虽然压缩后人很难看懂，但丝毫不影响浏览器执行。

尽管技术实现上有一些挑战，但云计算好处多多，你若经常上网定能有所体会。软件任何时候都是最新版本，信息保存在由专业人员管理的服务器上，而且容量不成问题。客户端数据可以实时备份，丢失信息的几率大大降低。任何文档可以只有一份，而不是在多个电脑里保存许多不一致的版本。共享文档很容易，即使基于同一份文档展开协作也易如反掌。云计算的价格呢，又十分低廉。

另一方面，云计算也带来了难以保护隐私和安全的问题。谁掌控着保存在云端的数据？谁可以读取它，什么情况下可以读取它？如果信息被意外泄露是否需要承担责任？谁有权强制公开数据？比如，在什么情况下，你的邮件服务商可以自愿或在法律诉讼的胁迫下向政府机关公开你的通信记录，或者在打官司时这么做？如果真发生了这种事，你会知道吗？这个问题的答案与你碰上了哪个国家的政府，以及你住在哪里有什么关系？如果你住在欧盟国家，那里关于个人数据隐私的规定相对严苛，而你的云数据保存在美国的服务器上，受《爱国者法案》（Patriot Act）管制，那该怎么办？

这些问题可不是凭空臆想出来的。身为大学教授，我自然有权读取通过邮件发过来的和保存在大学计算机中的学生信息，当然包括成绩，偶尔还有个人和家庭的敏感信息。我使用 Gmail 来收发邮件，使用 Google Docs 来保存和处理学生的成绩和信件是否合法？如果由于我个人方面的原因，导致这些信息可以被全世界看到，那会有什么后果？如果 Google 收到执法部门的传票，要求查询某个或一批学生的信息，那又该如何？我不是律师，也不知道答案。可是我担心这些事发生，所以我不使用云服务来保存学生档案和通信记录。如果我把这些信息都保存在单位配给我的计算机里，那至少由于单位管理疏忽或有错导致某些隐私泄露，我可以不承担责任。当然，如果是我自己把事情搞砸，那么无论数据保存在哪里，我都难逃其咎。

谁可以看你的邮件，有什么前提条件？这个问题部分涉及技术，部分涉及法律，而对法律方面的回答取决于你居住在哪个司法管辖区。在美国，我的理解是，如果你是一名公司员工，你的老板可以随便看你公司账号下的邮件，不用跟你打招呼，这不犯法。无论你的邮件是否与公司业务有关，老板都有这个权力。从理论上讲，你的邮件账号是老板为方便你开展工作才给你开立的，因此他有权确保这个便利工具被用于开展业

务，而且不违反公司规定和法律条款。

我的邮件通常没什么好看的，但如果校领导没有特殊原因随便就看，那即使他们有这个权力我也会非常不舒服。如果你是学生，你应该知道很多大学都把学生邮件当成个人隐私，跟普通的信件一样。根据我的经验，学生除了转发之外都不太常用他们的校园邮箱，他们把所有邮件都转到 Gmail，但这样就失去了本来会有的保护。

如果你像大多数人一样通过 ISP 或云服务来托管自己的邮件（比如使用 Gmail、雅虎、Hotmail、Verizon、AOL 和其他很多小一些的邮件服务），那隐私就只涉及你和它们。一般来说，这些服务商都公开宣称用户的邮件属于隐私。因此，除非接到传票，他们一般不会查看或泄露你的邮件。可他们也不会说明是否会拒绝那些拉大旗做虎皮，随随便便就以“国家安全”名义发来的传票。这些完全取决于服务商自己抗拒强权的意愿。反正美国政府已经多次试探舆论，想要更方便地查询邮件。9・11 之前的名义是打击有组织犯罪，之后则是反恐。来自政府的这种监控压力势必还会加大。

“云计算”这个术语也被用来称呼亚马逊等公司提供的服务。这类服务的基础是虚拟机，而任何人都可以使用亚马逊的计算机、存储系统和带宽。亚马逊的 EC2 或 Elastic Compute Cloud（弹性计算云），可以随着用户负载变化相应地增加或减少容量，而收费则是按实际使用量计价。亚马逊拥有的资源足以让任何个人用户瞬间扩展或收缩计算资源。

假设我想使用 EC2 开发一个服务，就叫“Space in the Cloud”吧。这个服务可以让我的用户把自己的任何信息上传到云端，并自动同步到他的所有设备上。我会给用户的数据加密，以确保不会被外人偷看到。为此，我为这个云服务写了一个程序，把它上传到亚马逊服务器，然后发布。现在，谁对用户隐私负有保密义务和责任呢？如果有人利用亚马逊服务中的一个 bug 偷走了用户文件，或我的付费用户的信用卡信息，或他们的纳税申报表，谁来负责？如果我的创业计划失败，那么谁能继续访问我已经收集的数据？如果政府找我配合调查可疑用户的行踪，我该怎么办？因为我有加密密钥，我的服务是提供加密保护的。

抛开隐私和安全问题，亚马逊或者其他提供商还应该负有什么责任？比如，2011 年 4 月亚马逊的部分服务宕机一天多时间，导致一些主流站点无法访问。如果因为一个配置错误导致这些站点的访问受阻，难以接受，那这些站点应该向谁索赔？标准的做法

是将这些内容写到服务协议中，但协议不能保证好的服务，它只能在事情发展到严重地步时为打官司提供一个依据。

维基解密（Wikileaks）这个站点曾公开过美国秘密外交电报，导致政府无地自容。2010年年底，亚马逊终止向维基解密提供服务，称对方违反服务条款，公布了自己不拥有或没有权力公布的内容，而且这样做也可能导致其他人受到伤害。亚马逊在政府并未提出要求的情况下自行采取了这一措施，但似乎这一举措源自政府暗中的压力。与此同时，一些财务金融机构，包括 PayPal、万事达卡、维萨卡和美国银行，都停止为维基解密提供付款服务，维基解密的一个域名解析服务提供商也停止了对它的服务。在此期间，美国政府要求 Twitter 提供一些维基解密涉案人员的所有记录，却遭到了 Twitter 抵制，Twitter 也没有答应替政府保密的请求。但任何时候都不能寄希望于任何组织的反抗。当然啦，世界上有“善意的”异议人士（如突尼斯和埃及），也有“恶意的”异议人士（如维基解密）。至少从哲学立场一致的角度来说，支持谁或打击谁并不容易抉择。

服务提供商对其客户有什么义务？什么时候应该拍案而起、据理力争，什么时候可能会向法律威胁或来自“权威人士”的声音低头？诸如此类的问题还可以提出一箩筐，但极少能给出清晰明确的回答。

11.6　小结

“举手之劳，无不可及”（Reach out and touch someone）曾是 1980 年代 AT&T 效果很好的一句广告语。今天，Web、电子邮件、社交网站、云计算，以及各种端到端的系统都可以实现这个愿景。有时候，这幅图景很美好。你可以在比以前大得多的圈子里交朋友，分享自己的兴趣爱好。然而，举手之劳也可以让你在世界面前暴露无遗，此时可不是人人都会只念你的好。垃圾邮件、欺诈、间谍软件、病毒、监视、冒名顶替、泄漏隐私信息，甚至遗失财产，种种不幸都会接踵而来。小心谨慎才是明智之举。”

第 12 章

结　束　语

本书到现在已经介绍了很多基本概念。读过这本书之后，你都学到了哪些东西？哪些在将来可能比较重要？五到十年之内，我们仍然需要解决哪些计算问题？哪些已经过时，或者说已经不重要了？

计算机与通信系统外在的形态和细节无时无刻不在变化。我们讲到的很多技术要点只是为了帮助大家理解事物背后的工作原理，这些地方不必过于深究。毕竟，大多数人容易理解具体的事例，不容易明白抽象的概念。而计算机和通信领域中抽象的概念实在是太多了。

对硬件部分来说，应该明白计算机的构成，它如何表示和处理信息，某些术语和数字的含义，以及计算机随着时间推移都有了哪些变化。

软件方面，关键是要知道怎么精确地描述任务，包括抽象的算法（同时考虑数据量的增加在多大程度上延长计算时间）和具体的程序。知道软件是由什么构成的，不同的编程语言怎么编写程序，程序怎么变成软件（通常是基于组件构建起来的），这样就能理解我们所用软件背后的秘密。但愿讲编程的那几章也会让你跃跃欲试，亲自动手写出一些代码来。

通信系统是无远弗届，无所不在的。重要的是理解其中的信息流动、谁能够查看这些信息，以及信息是如何得到控制的。协议，即系统之间交换信息的规则，也非常重要。协议的内容影响深远，由今天互联网中的身份认证问题可见一斑。

不少计算方面的概念对理解这个世界很有帮助。比如，我经常会区分逻辑结构与具体实现。这个根本问题有无数种表现方式。计算机就是一个典型的例子。无论其制造方式变化得有多快，其逻辑架构一直以来并没有太大不同。甚至可以认为，所有计算机的逻辑特征都是一样的，即它们都可以完成相同的计算。从软件角度看，代码作为一个抽象层，隔离了具体的实现。实现可以改变，而使用它们的代码可以不变。虚拟机、虚拟操作系统，甚至真正的操作系统都是利用接口来分离逻辑结构与具体实现的。想一想，编程语言也具备这个功能，有了它我们才可以跟计算机对话，就好像所有计算机都能听懂我们的话一样。当然，编程语言也是我们人类可以理解的。

计算机系统是设计上多方权衡、不断取舍的极佳范例，提醒我们设计中永远不可能处处如意，天下没有免费的午餐。桌面电脑、笔记本电脑、平板电脑、手机，同是计算设备，但它们在尺寸、重量、计算能力和成本等约束条件上，则分别作出了明显不一样的取舍。

计算机系统也是把大型、复杂系统切分成小型、易管理（可独立创建）组件的好例子。软件分层、API、协议和标准莫不如此。

我在导论中提到的“通用”对于理解数字技术同样重要。下面就来概括一下。

首先是**通用的数字信息表示**。化学有 100 多个元素，物理有十几个基本粒子。而数字计算机只有两个元素，0 和 1，其他一切都由此衍生出来。比特可用来表示任何信息，从最简单的真假、是否、对错之类的二元选择，到数字、字母，乃至一切事物。复杂的事物比如购物、浏览和手机历史中关于你生活的点点滴滴，则是由简单的数据项组成，后者又可以用更简单的形式来表示，如此往复，直到表示成一个一个的比特。

其次是**通用的数字处理器**。计算机是操作比特的数字设备。告诉处理器做什么的指令，被编码为比特，而且通常与数据保存在同样的存储器中。改变指令可以改变计算机行为，而这也正是计算机之所以成为通用机器的原因所在。比特的含义取决于上下文，一个人的指令可能是另一个人的数据。虽然有适合处理某种数据的特定技术存在，但复制、加密、压缩、错误检测等等操作全都可以在比特的层面上执行，与比特所表示的事物无关。运行通用操作系统的通用计算机取代各种专用设备的进程还将继续。未

来很可能出现根据生物计算原理设计的其他处理器，或许还会出现量子计算机。但是，数字计算机还会伴随我们很长时间。

第三是**通用的数字网络**，网络中从一个处理器传输到另一个处理器的数据和指令，同样也都是比特。互联网和电话网有可能融合为一个更通用的网络，恰似我们亲眼见证的计算和通信功能融合于今天的手机。互联网肯定会向前发展，至于是沿续其随心所欲、自由发展的特点，还是会受到商业和政府更多的制约，并没有明确的答案。（我想多半是后者，很不幸啊。）

最后，**通用的数字系统无所不在**。在整合多领域技术进步的基础上，数字设备向着小型、廉价和高性能的方向发展。某一领域的技术进步，比如存储密度，经常会影响到所有数字设备。

数字技术的核心限制和存在的问题仍然一如既往，对这些应该做到心知肚明 。技术给社会带来了无数积极的东西，但随之而来的则是很多新式难题或既有问题复杂化。以下是几个最重要的问题。

个人隐私不断受到来自商业和政府机构的威胁。我们个人的数据还会继续快速增长，而个人隐私保护可能会比现在的状况更不乐观。互联网从一开始就让人很容易匿名，特别是一些不良行为，要想匿名很容易。不过今天，即使是善意的行为，想要真正匿名都很困难。某些国家政府要求实名上网的提案非但不会给好人带来益处，反而给坏人提供了帮助和支持（仅此一点，就没有什么可取之处）。一言以蔽之，哪个国家的政府都希望自己国家的公民能够被轻易定位和监控，但又支持其他国家的异议人士匿名。大大小小的公司都渴望尽可能多地了解当前和潜在用户的情况。信息一旦上网，则永远无法消除。没有什么可行的办法能召回数据。

微型摄像头和麦克风，以及通过网页跟踪记录手机位置等**监控技术**一直在改进，而急遽降低的存储成本和处理成本，让完整地跟踪拍摄一个人的日常生活并将其数字化，变得越来越可行。如果要把你有生以来的所见所闻全部拍成视频保存起来，得需要多大的硬盘，这么大的存储空间花费几何？假设你今年 20 岁，那么需要的硬盘是 10 TB，购买这么大的存储空间在今天只要 500 美元，而且等你看到这些文字时一定还会更便宜。好啦，就算是把一个人一辈子都录成视频，多花 10 倍、20 倍也绝对足够了。

个人、公司和政府的**安全**也是始终存在的一个问题。不知道大家听到网络战之类的词汇后有什么感觉，反正可以确定的是，所有个人，还有稍大一些的公司，都潜在或经常受到某种网络攻击的威胁。

在一个没有任何成本就可以将数字内容复制无数份，而且没有任何成本就可以将这些副本传播到世界任何地方的环境下，**版权**是极难保护的。进入数字时代之前，音乐、电影和图书的制作和发行都需要成本，而且需要专业背景和特殊设备，因此这些创意产品的版权保护效果至少还是可以接受的。但这个时代已经过去了。传统版权保护及合法使用正在被授权许可和数字版权管理（DRM，Digital Rights Management）所取代，后者无法阻止真正的盗版，却给普通人造成了很多麻烦。该如何保护作者、作曲家、表演者、电影制作人和程序员的利益，同时确保他们的工作成果不会永久保密？

专利是一个严重的问题，而且越来越严重。随着越来越多的设备都内置了由软件控制的通用计算机，我们怎么保护创新者的合法权益，同时防止某些人利用手中过于宽泛或研究不足的专利来敲诈勒索？

资源分配，特别是像频段这样稀少但有价值的资源如何分配，始终是一个有争议的话题。既得利益者，比如手里已经握有资源的大型电信公司，占有先天的优势，它们可以通过金钱、游说和既有的网络效应来巩固自己的地位。

司法管辖权也是一个难题，因为信息可以流动到世界的任何一个角落。在一个地区合法的商业和社会行为，到了另一个地区可能就是违法的。法律体系远远没有跟上时代的步伐，矛盾已经随处可见。比如，美国的跨州征税问题，欧盟和美国之间数据隐私规定相互抵触的问题，再比如打专利或诽谤官司挑选法院时，原告会选择判决结果可能更有利于自己的司法管辖区，而不管犯罪现场或被告在哪里。

监管可能是最大的问题。政府希望控制自己的公民在互联网上的言论和行为，事实上所有媒体都越来越多地面临这种监管问题。公司希望控制自己的用户和竞争对手的行为。个人呢，当然愿意把政府和公司的触角限制在一定范围内，但长路漫漫，前途未卜。

最后，读者诸君务必牢记一点，无论今天的技术多么千变万化，人是不变的。无论从哪方面来看，现代的人类与几千年前的人类并没有太大区别。抚今追昔，历史上干好事的人有多大比例，今天也差不多；历史上干坏事的人有多大比例，今天同样也差不多。没错，社会、法律和政治都在适应技术变革，但这是个缓慢的过程，步调并不一致，而且世界不同角落的解决方案也各不相同。未来几年的发展变化一定会更加激动人心。希望这本书能够帮上你的忙，让你能够更加明智地预见、应对并积极影响其中一些不可避免的变化。

注 解

这一部分给出了本书参考资料的简要介绍（不敢保证没有遗漏），包括我自己喜欢而读者可能也会喜欢的一些书。当然，要调查或了解任何主题的基本事实，维基百科（Wikipedia）始终都是最好的资料来源，而搜索引擎在配合查找相关资料方面也发挥了重要作用。对于那些可以在网上轻易找到的资料，我都没有给出链接。

前言

- IBM 7094 大约配备 150 KB 内存，时钟速度为 500 KHz，售价约 300 万美元。参见：http://en.wikipedia.org/wiki/IBM_7090。
- Richard Muller 的《未来总统的物理课》（*Physics for Future Presidents*，Norton，2008）是一本非常好的科普书，它也启发我动笔写这本书。
- Hal Abelson, Ken Ledeen, Harry Lewis 的《数字迷城：信息爆炸改变你的生活》（*Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion*，Addison-Wesley，2008）。探讨了大量重要的社会和政治问题，特别是在互联网这个大背景下。这本书源自哈佛大学的一门类似的课程，其中很多东西放在普林斯顿的这门课上讲也是非常不错的。

导论

- James Gleick 的《信息简史》（*The Information: A History, A Theory, A Flood*，

Pantheon，2011）是一本关于通信系统的好书，主要围绕信息论之父克劳德·香农的故事展开。书中关于历史的讲解尤其引人入胜。

- James Essinger 的 *Jacquard's Web: How a Hand-Loom Led to the Birth of the Information Age*（Oxford University Press，2004）。从雅卡尔的织布机到巴贝奇、霍利里思（Hollerith）和艾肯（Aiken）。
- Doron Swade 的 *The Difference Engine: Charles Babbage and the Quest to Build the FirstComputer*（Penguin，2002）。Swade 也介绍了 1991 年营造的保存在伦敦科学博物馆的巴贝奇差分机。2008 年完成的另一台仿制品保存在加利福尼亚州山景城的计算机历史博物馆中。另见：http://www.computerhistory.org/babbage。

第一部分

- “如果说，只是举个例子，和声基调与音乐编排的内在关系容许如此表达和改编，那么可以说这台机器是精巧绝伦而又闪耀着科技光辉的音乐篇章，其复杂度或者说涵盖面之广前所未见。”摘自 1843 年 Ada Lovelace 对 Menabrea“分析机草图”（Sketch of the Analytical Engine）的译文及注解。
- Scott McCartney 的 *ENIAC: The Triumphs and Tragedies of the World's First Computer*（Walker & Company，1999）。
- Burks、Goldstine 和 von Neumann 的“Preliminary discussion of the logical design of an electronic computing instrument”，参见：http://www.cs.unc.edu/~adyilie/comp265/vonNeumann.html。
- 《傲慢与偏见》英文在线版：http://www.gutenberg.org/ebooks/1342。
- Charles Petzold 的《编码：隐匿在计算机软硬件背后的语言》（*Code: The Hidden Language of Computer Hardware and Software*，Microsoft Press，2000）。介绍了如何用逻辑门制作计算机，其内容比本书更加基础一到两个层次。
- Gordon Moore 的“Cramming more components onto integrated circuits”：ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf。
- “如果以 2 为基，则可以把得到的最小整数称为二进制数字（binary digit），或者用 J. W. Tukey 建议的那个更简洁的词，比特（bit）。”摘自克劳德·香农（Claude

Shannon）的“信息的数学理论”（A Mathematical Theory of Information，1948）。

- Seagate settlement：http://www.harddrive-settlement.com。
- 高德纳（Donald Knuth）的《计算机程序设计的艺术，卷 2：半数值算法》（*The Art of Computer Programming, Vol 2: Seminumerical Algorithms*，Addison-Wesley，1997）。
- 阿兰・图灵（Alan Turing）的“Computing machinery and intelligence”：http://loebner.net/Prizef/TuringArticle.html。The Atlantic 上有一篇关于图灵测试的文章，旁征博引，十分有意思：http://www.theatlantic.com/magazine/archive/2011/03/mind-vs-machine/8386。
- CAPTCHA 是一张公共域的图片，来源在此：http://en.wikipedia.org/wiki/File:Modern-captcha.jpg。
- Andrew Hodges 维护的图灵的主页：http://www.turing.org.uk/turing。Hodges 是权威人物传记《阿兰・图灵传：如谜的解谜者》（*Alan Turing: The Enigma*，Burnett Books，1983）的作者。
- ACM 图灵奖：http://awards.acm.org/homepage.cfm?awd=140。

第二部分

- Steve Lohr 的《软件简史》（*Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts—the Programmers Who Created the Software Revolution*，Basic Books，2001）。
- Kurt Beyer 的《优雅人生：格雷斯・霍珀和信息时代的创新》（*Grace Hopper and the Invention of the Information Age*，MIT Press，2012）。Hopper 是一位举足轻重的人物，一位有巨大影响力的计算机先驱，她在 79 岁退休时，创下了美国最年长海军军官的纪录。她在演讲时众所周知的一个惯用姿态是伸出双手比划出一步宽，说：“这是一纳秒。”
- NASA 火星气象卫星报告：ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf。
- bug 照片是一张公共域的图片，来源如下：http://en.wikipedia.org/wiki/File:H96566k.jpg。

- 美国最高法院决定确认《1998 桑尼·博诺版权期限延长法案》（被人笑称为《米老鼠保护法案》）不违宪：http://en.wikipedia.org/wiki/Eldred_v._Ashcroft。
- Larry Lessig 的《代码 2.0：网络空间中的法律》（*Code 2.0*，Basic Books，2005）。这本书也有一个按照知识共享许可发布的在线版：http://codev2.cc。
- Amazon 1-click 专利：http://www.google.com/patents?id=O2YXAAAAEBAJ。
- 法院的事实裁定书，第 154 段（1999）：http://www.justice.gov/atr/cases/f3800/msjudgex.htm。2011 年，当对微软的承诺做了最后一次监督审查之后，这个案子就结束了。
- 一个很实用的 JavaScript 学习网站：http://w3schools.com。

第三部分

- Gerard Holzmann 和 Bjorn Pehrson 的 *The Early History of Data Networks*（IEEE Press，1994）。关于视觉电报详尽而又有趣的历史。
- 视觉电报图是一张公共域的图片，来源在此：http://en.wikipedia.org/wiki/File:Chappe_telegraf.jpg。
- Tom Standage 的 *The Victorian Internet: The Remarkable Story of the Telegraph and the Nineteenth Century' On-Line Pioneers*（Walker，1998）。愉快而奇妙的阅读体验。
- 承蒙 John Wait 惠赐旋转拨号盘电话的照片。
- Edward Felten 的“网络中立性的基本要素”（Nuts and Bolts of Network Neutrality）：http://citp.princeton.edu/pub/neutrality.pdf。
- 承蒙 Gerard Holzmann 惠赐 RJ45 连接头照片。
- Guy Klemens 的 *Cellphone: The History and Technology of the Gadget that Changed the World*（McFarland，2010）。关于手机发展的详尽历史和技术真相，有些地方比较枯燥，但大多数内容都很好理解。这本书为我们已经司空见惯的复杂系统描绘出了一幅幅清晰的画面。
- 关于鸟类运送信息的 RFC：http://tools.ietf.org/html/rfc1149。
- SMTP 课程：http://technet.microsoft.com/en-us/library/bb123686.aspx。
- 美国司法部没收的域名：http://viewdns.info/research/inside-the-dojs-domain-name-graveyards。

- 微软“关于安全的 10 个不变的法则”：http://technet.microsoft.com/en-us/library/cc722487.aspx。
- RSA 关于 RSA 攻击的公开声明：http://www.rsa.com/node.aspx?id= 891。
- Simon Singh 的 *The Code Book*（Anchor，2000）。一本能让普通读者愉快地了解加密历史的书。写 Babington 阴谋（企图把苏格兰皇后扶上王位）那一段太精彩了。
- 卡内基梅隆大学计算机教授 David Touretzky 维护着一个网页，里面有关于 DeCSS 的各种介绍：http://www.cs.cmu.edu/~dst/DeCSS/index.html。
- Alice 的 *Bob and Eve*：http://xkcd.com/177。
- RSA 因式分解挑战：http://www.rsa.com/rsalabs/node.asp?id=2092。
- Bruce Schneier 的《网络信息安全的真相》（*Secrets and Lies: Digital Security in a Networked World,* Wiley, 2000）。描写了对安全和隐私的威胁及破坏，非常适合非专业人士阅读。Schneier 关于安全的博客：http://www.schneier.com。
- 谷歌最初的论文：http://infolab.stanford.edu/~backrub/google.html。这个系统最初叫做“BackRub”。
- Ken Auletta 的《被谷歌》（*Googled: The End of the World As We Know It*，Penguin，2009）。
- Steven Levy 的 *In the Plex: How Google Thinks, Works, and Shapes our Lives*（Simon & Schuster，2011）。
- Siva Vaidhyanathan 的 *The Googlization of Everything*（University of California Press，2011）。
- Bala Krishnamurthy 和 Craig Wills 谈跟踪：http://www.research.att.com/~bala/papers/www09.pdf。Krishnamurthy 的网站上有大量关于隐私的实证研究。
- 位置隐私：http://www.eff.org/wp/locational-privacy。电子前线基金会（The Electronic Frontier Foundation）的网站 http://eff.org 包含很多有价值的隐私及安全政策方面的内容。
- Carter Jernigan 和 Behram Mistree 的“Gaydar: Facebook friendships expose sexual orientation”：http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/2611/2302。
- Evgeny Morozov 的 *The Net Delusion: The Dark Side of Internet Freedom*（Public

Affairs，2011）。那些让我们更加方便地相互联系的技术和系统，也让当局跟踪调查我们更加容易。

- Clayton Christiansen 的《创新者的窘境》（*The Innovator's Dilemma*，Harper，1997）。阐述了那些潜移默化间导致现有技术和公司失败的破坏性手段，从低价到山寨。
- Jonathan Zittrain 的 *The Future of the Internet—And How to Stop It*（Penguin，2008）。互联网正从一个完全开放的、每个人都能为它添砖加瓦的环境，向着一个由供应商主导的封闭的"设备化"系统发展。

词　汇　表

这里给出了本书中出现的重要术语的简明定义或解释，主要是那些字面上看虽然普普通通，但却有着特殊含义，而且使用频率比较高的一些词汇。

计算机和互联网等通信系统要处理大量数据，这些数据经常以不同的单位表示。下表列出了本书中出现的所有单位，以及国际单位制（International System of Units，SI）中的其他一些单位。随着技术的进步，我们还可能看到更多表示大数的单位。这张表里也列出了与每个单位最接近的 2 的幂。10^{24} 的误差只有 21%，换句话说，2^{80} 约等于 1.21×10^{24}。

国际单位制名称	词头符号及英文名	10的幂	常用名	2的幂
尧[它]	Y（yotta）	10^{24}		2^{80}
泽[它]	Z（zetta）	10^{21}		2^{70}
艾[可萨]	E（exa）	10^{18}	百京	2^{60}
拍[它]	P（peta）	10^{15}	千兆	2^{50}
太[拉]	T（tera）	10^{12}	兆	2^{40}
吉[咖]	G（giga）	10^{9}	十亿	2^{30}
兆	M（mega）	10^{6}	百万	2^{20}
千	k（kilo）	10^{3}	千	2^{10}
		10^{0}		2^{0}
毫	m（milli）	10^{-3}	千分之一	2^{-10}
微	μ（micro）	10^{-6}	百万分之一	2^{-20}
纳[诺]	n（nano）	10^{-9}	十亿分之一	2^{-30}

（续）

国际单位制名称	词头符号及英文名	10的幂	常用名	2的幂
皮[可]	p（pico）	10^{-12}	兆分之一	2^{-40}
飞[母托]	f（femto）	10^{-15}		2^{-50}
阿（托）	a（atto）	10^{-18}		2^{-60}
仄（普托）	z（zepto）	10^{-21}		2^{-70}
幺（科托）	y（yocto）	10^{-24}		2^{-80}

3G　第三代（third generation），描述 iPhone 和 Android 等智能手机带宽的一个不准确的术语。

802.11　无线网络系统的标准，常见于笔记本电脑和家中路由器，也叫 Wi-Fi。

AES　即 Advanced Encryption Standard（高级加密标准），是应用非常广泛的一种密钥加密算法。

AM　即 Amplitude Modulation（调幅），给无线电信号添加语音或数据信息的一种机制，常见于广播电台的 AM 广播。

API　即 Application Programming Interface（应用程序编程接口），对库或其他软件集所提供服务的描述。例如，Google Maps API 描述了如何通过 JavaScript 控制地图的显示。

ASCII　即 American Standard Code for Information Interchange（美国信息交换标准代码），涵盖了英文字母、数字和标点符号的 8 位编码方案。

BitTorrent　有效分发大型热门文件的端到端协议，下载者同时也上载。

bug　程序中的差错。

Chrome　谷歌公司的浏览器。Chrome OS 是只支持一个浏览器的操作系统。

cookie　服务器发送的文本文件，由浏览器保存在本地计算机上，下一次访问时再由浏览器发送给服务器，普遍用于跟踪用户浏览的网站。

CPU　即 Central Processing Unit（中央处理单元）。参见处理器。

DES 即 Data Encryption Standard（数据安全标准），第一种广泛使用的数字加密算法，后被 AES 取代。

DMCA 即 Digital Millennium Copyright Act（数字千年版权法案，1998），美国用于保护数字作品版权的法律。

DNS 即 Domain Name System（域名系统），把域名转换成 IP 地址的互联网服务。

DSL 即 Digital Subscriber Loop（数字用户环路），通过电话线发送数字信息的一种技术，与有线电视电缆上网类似。

EULA 即 End User License Agreement（最终用户许可协议），通常是小字号打印或显示的长长的法律文本，用于限制用户对软件或其他数字信息的使用方式。

Flash Adobe 公司用于在网页上显示视频和动画的软件。

FM 即 Frequency Modulation（调频），通过改变无线电信号的频率来发送信息的一种技术，比如广播电台的 FM 广播。

GNU GPL 即 GNU General Public License（GNU 通用公共许可），一种保护开源代码的版权许可，要求必须开放源代码，以防被变成私有的。

GPS 即 Global Positioning System（全球定位系统），使用卫星发送的时间信号计算物体在地球表面的位置，是一种单向通信机制。汽车导航仪等 GPS 设备不会向卫星发送信号。

GSM 即 Global System for Mobile Communications（全球移动通信系统），全世界 80%的人在使用的手机通信机制，也是美国采用的两种手机通信机制之一。

HTML 即 Hypertext Markup Language（超文本标记语言），用于描述网页的内容和格式。

IC 即 Integrated Circuit（集成电路），在平滑的材料表面上加工而成，安装在密封的外壳中，与电路的其他设备相连。大多数数字设备的主体都是 IC。

ICANN 即 Internet Corporation for Assigned Names and Numbers（互联网名称与数字地址分配机构），负责分配互联网资源的机构，必须保护唯一性，比如域名或协议编号。

IP 即 Internet Protocol（互联网协议），规定如何通过互联网发送数据包的一个基础协议。

IP 地址 互联网协议地址，是唯一标识互联网上每台计算机的数字地址，类似于电话号码。

ISP 即 Internet Service Provider（互联网服务提供商），提供互联网接入服务的实体，比如大学、有线电视或电话公司。

JavaScript 网页中使用的编程语言，可以控制页面的显示效果，也被广泛用于跟踪用户。

JPEG 即 Joint Photographic Experts Group（联合图像专家组），一种压缩和表示数字图像的算法。

Linux 开源的类 Unix 操作系统，广泛用于服务器。

MP3 一种压缩和表示数字音频的算法，是 MPEG 视频压缩标准的一部分。

MPEG 即 Moving Picture Experts Group（运动图像专家组），一种压缩和表示数字视频的标准算法。

PDF 即 Portable Document Format（便携文档格式），最初由 Adobe 发明的一种表示文档的标准。

RAM 即 Random Access Memory（随机访问存储器），计算机中的主存储器。

RFID 即 Radio-Frequency Identification（无线射频识别），一种功率很低的无线通信机制，可用于电子门禁、宠物植入芯片等。

RGB 即 Red, Green, Blue（红，绿，蓝），是计算机显示器用三基色合成颜色的标准方式。

RSA 使用最广泛的公钥加密算法。

SDK 即 Software Development Kit（软件开发工具包），帮助程序员为某种设备或环境（比如手机或游戏主机）编写程序的工具集合。

TCP 即 Transmission Control Protocol（传输控制协议），使用 IP 创建双向通信流的协议。TCP/IP 是 TCP 和 IP 的组合。

Unicode 对世界上所有书写系统涉及的全部字符进行编码的标准方式。

Unix 由贝尔实验室开发的一个操作系统，今天的很多操作系统都是由它衍生而来的。Linux 是一个看起来与之相像，也提供相似功能，但实现不同的另一个操作系统。

URL 即 Uniform Resource Locator（统一资源定位符），网址格式的标准，比如：http://www.amazon.com。

USB 即 Universal Serial Bus（通用串行总线），可插拔式计算机外围设备（如外部磁盘驱动器、数码相机、显示器和手机）的标准插头。

VoIP 即 Voice over IP（IP 电话），利用互联网传送语音的一种方法，经常要借助常规的电话系统。Skype 是一个常用的 VoIP 服务。

Web 服务器 托管网站或 Web 应用的服务器。

Wi-Fi 即 Wireless Fidelity（无线保真），802.11 无线通信标准的市场宣传用语。

包 以特定格式构成的一段信息，比如 IP 包，可以大致理解为一个标准的集装箱货柜。

比特（位） 表示开或关这种二选一状态的二进制数（0 或 1）。

编程语言 用于表达计算机操作的各种符号，最终会被转换成比特加载到 RAM 中，比如汇编语言、C、C++、Java 和 JavaScript。

编译器 负责把高级语言（如 C 或 Fortran）编写的代码翻译成低级语言（如汇编语言）代码的程序。

变量 内存中用于保存信息的一个位置。变量声明定义变量的名字，也可以同时给它赋一个值，这个值叫变量的初始值，将来也可以重新赋予它一个相同数据类型的值。

标准 关于某物如何运行或如何构建或控制它的正式规范或描述，精确到能够保障互用性和相互独立地实现。比如 ASCII 和 Unicode 字符集、USB 的插头和插槽，以及编程语言的详细定义。

表示 描述如何以数字形式表达数据的一种通用说法。

病毒 恶意程序，感染计算机。病毒与蠕虫不同，需要有人帮助才能传播到其他系统。

操作系统 控制计算机资源的程序，资源包括 CPU、文件系统、设备和外部连接。比如 Windows、Mac OS X、Unix、Linux。

插件 可以在浏览器中运行的程序，比如 Flash、Quicktime 和 Silverlight。

程序 一组能让计算机完成某个任务的指令，用编程语言编写。

处理器 计算机中执行算术和逻辑运算，同时控制其他计算机部件的部分，也称 CPU。英特尔和 AMD 处理器在笔记本电脑中使用得最多，ARM 处理器主要用在手机中。

代码 用编程语言编写的程序文本，也说源代码。一种编码，比如 ASCII 编码。

带宽 对通信路径传输信息速度有多快的描述，比如电话调制解调器为 56 Kbit/s，而以太网为 100 Mbit/s。

电缆调制解调器 用于通过有线电视网络发送数字信息的设备。

端到端 不同的客户端之间交换信息，与客户端–服务器相对，是一种对称、平等的关系。 最常用于文件共享网络。

对数 对于一个数值 N，就是某个基数要得到它所需要的幂指数。在本书中，基数是 2，对数是整数。

恶意软件 不怀好意或图谋不轨的软件。

二次增长 以某个数值的平方为增长率，比如选择排序的运行时间/次数会随着要排序的项数增加呈二次增长。

二分搜索 一种搜索有序列表的算法，反复把要搜索的部分分成同样大小的两半。

二进制 表示只有两种状态或可能的值，也指以 2 为基的数系中的二进制数值。

防火墙 控制或屏蔽进出计算机的网络连接的软件。

服务器 根据客户端的请求提供数据访问服务的一台或多台计算机，比如搜索引擎、购物网站和社交网站等。

复杂性 对某个计算任务或算法的难度的度量，用处理 N 个数据项所花的时间/次数（比如 N 或 $\log N$）表示。

跟踪 记录上网用户访问过哪些网站，以及做了些什么。

固态硬盘 使用闪存的存储装置，是磁盘驱动器的替代品。

光纤 极为纯净的玻璃纤维，用于在较大的地理范围内传输光信号，信号中编码了数字信息。距离最远的数字网络都是通过光纤电缆连接的。

函数 计算机程序中的一种构造，可以执行某种特定的计算任务，比如计算一个数的平方根或弹出一个对话框（JavaScript 中的 `prompt` 函数）。

缓存 本地存储空间，可以加快重新访问近期访问过的数据的访问速度。

汇编器 负责把 CPU 指令系统中的指令翻译成比特，以便直接加载到计算机内存中的程序。汇编语言是对应的编程语言。

基站 把无线设备（手机、计算机）连接到某个网络（电话网、计算机网络）的无线电发射装置。

架构 粗略地表示计算机程序或系统的组织或结构的一个词。

间谍软件（后门程序） 将自己所在计算机的情况传回自己主机的软件。

僵尸，僵尸网络 被坏人控制的运行恶意程序的计算机。僵尸网络是由同一个人控制的一批僵尸计算机。

接口 用于描述两个独立实体之间界限的一个含糊的通用词汇。参见 API 的编程接口。接口有时也会被称为界面，即直接与人类交流的那部分计算机软件。

解释器 为计算机解释指令的程序，无论是实体的还是虚拟的，总之具有解释功能。浏览器中的 JavaScript 程序就是由解释器来处理的。

开源 可以自由使用的（以高级语言编写的）源代码，通常按照 GNU GPL 等类似开源许可发布。

客户端 一个程序，通常是浏览器，可以向服务器发送请求，比如，我们常说客户端–服务器。

库 一组相关的软件组件，可以用于搭建程序。比如 JavaScript 提供的用于访问浏览器的标准函数。

扩展 附加安装在浏览器中的小程序，为用户提供额外的功能或便利。比如用于管理隐私的 Adblock 和 NoScript 都是扩展。

蓝牙 小范围、低功率的无线通信标准，适用于手机、游戏机等。

浏览器 大多数人都在使用的一种上网软件，比如 Internet Explorer、Firefox 或 Safari。

路由器 网关的别称，即把信息从一个网络发送到另一个网络的计算机。

模拟 利用物理性质成比例变化的特性来表示信息的一种方法，比如温度计中的液位，与数字相对。

模拟器 模拟某种设备或系统的（行为类似的）程序。

目标码 二进制形式的指令或数据，可以直接加载到 RAM，是编译和汇编的结果。与源代码相对。

目录 就是文件夹。

内核 操作系统的核心，负责控制运行和资源分配。

频段 手机、广播电台等系统或设备使用的频率范围。

平台 描述软件系统的一个含糊的词汇，比如操作系统，可以作为其他服务的基础。

驱动程序 控制特定硬件（如打印机）的软件，通常由操作系统按需加载。

蠕虫 恶意程序，感染计算机。蠕虫与病毒不同，不需要帮助就能传播到其他系统。

闪存 一种集成电路存储技术，不耗电也能保存信息，广泛用于数码相机、手机、USB 记忆棒，是磁盘驱动器的一种替代品。

社交工程 欺骗受害人使其透露某些信息的手法，通常是伪装成与受害者都认识某个人，或同在一家公司工作。

声明 编程语言中的构造，用于定义程序中某一部分要使用的名字和属性，比如声明计算过程中用于存储信息的变量。

十六进制 以 16 为基的数制，常见于 Unicode 编码表、网址（URL）、颜色值。

数字 表示信息的离散的数值，与模拟相对。

搜索引擎 必应（bing.com）或谷歌等公司收集网页并提供查询服务的服务器。

算法 对计算过程精确而完整的说明，与程序相比它又是抽象的，不能通过计算机执行。

特洛伊木马 计算机病毒，宣称要做什么，但实际上却做另一件事，而且往往是恶意的。

调制解调器 包含调制器和解调器，可以实现比特和模拟信号（如声音）之间相互转换的设备。

图灵机 抽象的计算机，由阿兰·图灵发明，可以执行任意数字计算任务。通用图灵机可以模拟其他图灵机，因而可以模拟任何数字计算机。

外围设备 与计算机相连的硬件设备，如外接磁盘、打印机、扫描仪，等等。

网关 连接计算机网络的一种特殊计算机，也常被称为路由器。

网络钓鱼，鱼叉式网络钓鱼攻击 伪装成与受害人有某种关系，利用电子邮件骗取个人信息。鱼叉式网络钓鱼的定位更准确。

网络中立性 一个普适的原则，即互联网服务提供商应该对所有通信流量一视同仁（过载的情况下除外），不能出于经济或技术的原因而采取区别对待的策略。

网页信标　一个很小而且通常是看不到的图片，用于记录某个网页是否已经被下载过了。

微芯片　芯片或集成电路的另一种说法。

文件夹　一个文件，用于保存其他文件和文件夹信息，比如大小、日期、权限和位置。

文件系统　操作系统中负责组织和访问磁盘及其他存储介质中数字信息的部分。

无线路由器　把无线计算机连接到有线网络（通常是以太网）的无线电设备。

系统调用　操作系统将自己的服务提供给程序的一种机制。系统调用与函数调用非常类似。

像素　即 Picture element（图元），构成数字图像的一个点。

协议　关于系统间如何交互的规范，互联网中最常见，它的运行依赖于一大批网络间交换信息的规范。

芯片　小型电子电路，在平滑的硅表面上加工而成，安装在陶瓷外壳中，也称集成电路或微芯片。

虚拟机　模拟计算机的一种程序，也称为解释器。

虚拟内存　软件与硬件结合起来给人一种 RAM 没有限制的假象。

循环　重复执行一组指令的程序段，无限循环会重复很多很多次。

压缩　减少数字信息占用的比特数，比如将数字音乐压缩成 MP3。

以太网　最常见的局域网络技术，大多数家庭和办公室的无线网都是以太网。

应用，应用程序　能够完成某些任务的程序或程序套件，比如 Word 或 iPhoto。一般称手机上的应用程序（如日历、游戏等）为“应用”。

硬盘　在可以旋转的磁性盘面上存储数据的设备，也叫硬盘驱动器。

域名　对于连接到互联网的计算机所给出的层次化的命名方案，比如 www.msnbc.com。

源代码 用编程语言编写的程序员能够看懂的程序文本，最终将被编译为目标码。

云计算 在服务器上执行计算，数据在保存在服务器上，可以代替桌面应用程序。电子邮件、在线日历和图片分享网站都是例子。

证书 一段加密数据，可用于验证网站的真实性。

知识产权 创造或发明行为的产品，受版权法和专利法保护，包括软件和数字产品。

指数级 每次都按照一个固定比例增长，比如每年翻一番或（结果相同的）每月增长6%。通常用这个词泛指“快速增长”。

智能手机 iPhone 和 Andriod 手机之类的具有下载和安装应用能力的手机。

注册商 （从 ICANN）得到授权向个人和公司出售域名的公司，如美国人都知道的GoDaddy。

字节 8 比特（位），足够存储一个字母、一个较小的数值或一个更大量的一部分，是现代计算机中的一个基本单位。

总线 用于连接电子设备的一组线。参见 USB。

索　引

A

Ad Exchange, 218
ADSL, 137
AdWords, 216
AES, 203
Android, 95
ARPANET, 150
ASCII, 30
按点击收费, 215
按页面浏览量收费, 215

B

BASIC, 76
版权, 85
编译器, 73
标准, 89

C

C, 76
C++, 76
CAPTCHA, 50
CDMA, 145
Chrome OS, 111
COBOL, 75
cookie, 217
CTSS, 94
操作系统、系统调用、驱动程序和应用程序之间的关系, 100
超级计算机, 49
处理器, 13
传感器阵列, 26
传输控制协议, 159
次级存储, 15

D

DES, 203
DRM, 85
DSL, 137
打地理标签, 216
带宽, 135
等待或延迟, 135
调制, 136
抖动, 135

E

EDSAC, 72

F

FAT, 107

FORTRAN, 75
分布式计算, 49
分布式拒绝服务攻击, 198
分支, 42

G

GNU, 90
GPL, 90
GPS, 143
GSM, 145
感知编码, 175
固态盘, 16

H

HTTP 协议是“无状态”的, 184
函数, 81
函数库, 81
互联网协议, 159
汇编器, 72

I

IC, 20
iOS, 95
IP 包, 151
IP 地址, 152

J

Java, 77
JavaScript, 78
JPEG, 29
基站, 144
集成电路, 20, 21, 22
间谍软件, 196
僵尸, 196
僵尸网络, 196
解调, 136
净室开发, 86

K

开放源代码, 90

L

Linux, 97
来抢劫我吧, 224
流水线, 46
留声机, 26
旅行推销员问题, 67

M

Mac OS X, 95
MPEG, 29
蛮力攻击, 203
密纹唱片, 26

N

NP 问题, 66
内存, 14, 15
内核, 111

O

Objective-C, 77

P

PageRank, 214

Q

奇偶校验码, 176

R

RAID, 108
RFID, 143
肉鸡, 196

S

SD 闪存卡, 107
SQL 注入, 197
商业秘密, 85
社交图谱, 225
随机访问存储器, 14

T

TCP/IP, 159
特洛伊木马, 191
跳转, 42
图灵测试, 50
图灵机, 50

U

Unicode, 30
Unix, 94
URL, 181
USB 闪存盘, 107

V

VPN, 198

W

Windows, 95
WPA, 147
维基解密, 230
位置隐私, 225
无损压缩, 174

X

系统编程, 76
协议, 153
信程, 135
许可, 87

Y

亚马逊的“一键购买”（1-click）专利, 86
亚马逊的 EC2, 229
沿街扫描, 148
硬盘, 15, 17
有损压缩, 174
鱼叉式网络钓鱼, 195
语法, 117
语义, 117
域名系统, 152
云端, 226

Z

字典攻击, 203
总线, 13
震网, 191
证书, 207
指数级增长, 66
主存储器, 14
专利, 86

版权声明





李松峰

图灵QA部主任。2006年起投身翻译，出版过译著20余部，包括《JavaScript高级程序设计》、《简约至上》等畅销书。2008年进入出版业，从事技术图书编辑和审稿工作。

他于2007年创立知识分享网站“为之漫笔”（cn–cuckoo.com），翻译了大量国外经典技术文章。2012年下半年创立“A List Apart中文版”站点（alistapart.cn），旨在向中文读者译介这一国际顶级Web设计与开发杂志。他经常参加技术社区活动，曾在w3ctech 2012 Mobile上分享“Dive into Responsive Web Design”。2013年1月应邀在金山网络分享“响应式Web设计”，2013年3月应邀在奇虎360分享“JS的国”。



徐建刚

网名adoal，山东青岛人，1975年生，就职于浙江大学图书与信息中心，从事图书馆信息化与数字图书馆研究工作。有丰富的服务器系统运维经验，对互联网领域常见的开源操作系统、电子邮件服务器、目录服务器等软件有深入的了解。熟悉多种主流编程语言，尤喜Python。热心开源软件推广，活跃在水木等国内技术社区。此外，他还是浙江大学开源软件镜像站的负责人和主要维护者。



世界是数字的

“对计算机、互联网及其背后的奥秘充满好奇的人们，这绝对是一本不容错过的好书。”

——谷歌常务董事长　埃里克·施密特

“Kernighan这本书是写给普通人看的，写得简直棒极了！这本书是他在哈佛休假期间写的，因此我有幸先睹为快。Kernighan的文字简明易懂、妙趣横生，又发人深省。不管是专业人士还是普罗大众，他的真知灼见都会让大家有所思，有所得。”

——哈佛大学计算机科学教授、哈佛学院前院长、盖茨和扎克伯格导师　哈瑞·刘易斯

“我上高中的儿子对计算机本来一窍不通，但这本书让他觉得非常有意思。我自己看过作者与C语言之父合著的*The C Programming Language*，那可是当时的C语言标准！如今，作者乐意跟不懂技术的人聊聊我们生活中的计算机，这真是太好了。”

——美国计算机协会会士、麻省理工学院计算机科学与人工智能实验室研究员　大卫·卡尔格

“十分荣幸能在普林斯顿大学亲耳聆听作者针对非计算机专业学生开设的计算机课程，课上使用的就是这本书。这本书讲到了数字世界的所有基本常识，读起来一点儿都不像课本，倒像一本故事书。这门课结业之后，我把书送给了老妈。现在，老妈已经是她办公室的计算机专家了！不管你有没有计算机背景，就算你是位计算机高手，我也会向你推荐这本书！”

——亚马逊读者



封面设计 自在书装设计 83720326@qq.com

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育　@图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　科普读物/互联网

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-31875-6
定价：49.00元

欢迎加入

图灵社区

电子书发售平台

电子出版的时代已经来临，在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版的梦想。你可以联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者，这极大地降低了出版的门槛。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果有意翻译哪本图书，欢迎来社区申请。只要通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

读者交流平台

在图灵社区，读者可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。欢迎大家积极参与社区开展的访谈、审读、评选等多种活动，赢取银子，可以换书哦！